APL+Win v10

New System Features

Robustness, Control Flow, Error Handling, and Debugging

Copyright

While every attempt has been made to ensure that the information in this document is accurate and complete, some typographical errors or technical inaccuracies may exist. APLNow, LLC does not accept responsibility for any kind of loss resulting from the use of information contained in this document.

This page shows the publication date. The information contained in this document is subject to change without notice. Any improvements or changes to either the product or the document will be documented in subsequent editions.

APL+Win 10

This edition published 2010.

APL+Win v10.0 is the most substantial release of the system in several years. It is filled with significant enhancements including much faster performance, larger workspace size, and many new features oriented toward making programming more productive and applications more reliable.

What's New?

- System speed and memory capacity have increased dramatically. This is the fastest APL+Win system ever! Some applications now run in as little as half the time they required on the previous version. Maximum workspace size has grown from about 1.6 GB previously to 3.6 GB (on Win64 systems) and 2.6 GB (onWin32 systems with 4GT enabled).
- Support for bigger sized arrays in the workspace. In the previous version, the array was just limited to 214,748,352 elements. In APL+Win v10.0, the array may be as big as 2,147,483,647 elements (10x bigger) or up to 2 GB in size, whichever is smaller.
- Good programming practices are encouraged via separate debug and release execution modes. Debug mode can be used during development whenever correctness testing is critical, whereas release mode can be used for application deployment whenever speed is critical. The *:DEBUG*, *:TRACE*, *:ASSERT*, and *:IFDEBUG* statements execute only during debug mode. When release mode is enabled, inner-statement flow is optimized to completely avoid execution of debug statements, making them logically invisible with zero execution cost as if they were comments.
- In addition to debug mode, a rudimentary testing infrastructure facilitates semi-automatic self-testing. Tests coded in *:TEST … :ENDTEST* blocks are logically invisible during normal execution. Test blocks may be defined directly inside the functions they test or elsewhere.
- The *TraceLog* feature can capture execution history of up to 134 million statements with less than 2% performance penalty. This makes it possible to explore control flow errors without modifying the code or slowing it down enough to make bug mutation likely.
- A crash recovery mechanism supports automatic application restart in the event of failure. It creates a crash log file containing information useful for debugging and a MiniDump file, which APL2000 can execute with a post-mortem debugger, to help diagnose and fix bugs in the APL+Win system or determine if the crash is happening in a third party component.
- Most debugging and tracing features can be re-enabled after application release via configuration file settings, manually edited by the client or via debugging options on the application's menus. This can help diagnose bugs that only occur at client sites and cannot be reproduced in the lab.
- Complex logical calculations can now be concisely coded via the experimental Inline Control Sequences (ICS) feature using *:AND*, *:OR*, *:THEN*, *:ELSE*, and *:CHOOSE* keywords embedded in expressions. They are evaluated progressively similar to the `&&`, `||`, and `?` operators found in C, C++, C#, JavaScript, etc.
- The *:RETURN* statement allows a result expression as an optional argument. When an argument is coded, its value is returned as the function's result, shortcutting the normal two step process of assigning a result variable and then returning. Without an argument, *:RETURN* behaves normally.
- The *:RETURNIF*, *:LEAVEIF*, and *:CONTINUEIF* statements do conditional exit/continue with a single statement rather than a three statement *:IF … :ENDIF* sequence surrounding a *:RETURN*, *:LEAVE*, or *:CONTINUE* statement. *:RETURNIF* also allows an optional result value argument.
- The *:TRY* statement has a new mechanism to pop errors that occur in called functions back to its handling context regardless of the ambient *⎕ELX* setting. Its *:CATCH* clause supports wildcard pattern matching, which is more concise and efficient than *:CATCHIF*. Its *:FINALLY* clause defines an always-executed block of code, which runs no matter how the *:TRY* statement is exited.
- The *:TRYALL* statement provides resume-on-next-line error handling, which is more convenient in some cases than traditional *:TRY … :CATCH* or *⎕ELX* error handling. It is similar to the

effect of setting `⎕ELX←'→⎕LC+1'` but without the problems doing so could create in called functions.

- The `:SELECT` statement supports a wildcard pattern matching `:LIKE` clause, which uses the same notation as `:CATCH`, as an alternative form of `:CASE` and `:CASELIST` clauses. It also supports a `:NEXTCASE` statement to flow from one case into the next without branching.
- The `:FOR` statement now supports multi-variable strand notation to the left of `:IN` keyword.
- `⎕VALENCE`, `⎕MONADIC`, and `⎕DYADIC` functions to simplify testing of valence and `⎕NOVALUE` variable to explicitly indicate no value.
- The `⎕THROW` function is similar to `⎕ERROR` except it throws the error in the context of the calling function rather than exiting from the function before throwing it. This allows errors thrown in a `:TRY` block to be handled locally rather than existing from the function.
- The `⎕LOG` function routes output to a debug log file separate from the APL session and/or writes events to the Windows Event Log. The `:TRACE` statement implicitly calls `⎕LOG` in debug mode.
- The debugger can now stop for errors at their point of origin and/or handling. This is useful in general but especially for errors in `:TRY` statements, which were previously impossible to debug.
- Unicode clipboard support simplifies inter-user communication, documentation, creation of web pages containing APL code, and helps promote APL in the non-APL community.
- The runtime system now loads development and runtime workspaces; workspaces saved with either `)save` or `)rsave`.
- `⎕WGIVE` optimized to execute with less overhead.
- Tabs are now treated syntactically the same as blank spaces in APL code when not in quotes.
- New "Event Stop Settings" in Code Walker for improved debugging error handling code.
- The `)TEST` system command and the `⎕TEST` system function are used to execute statements that are contained inside of the `:TEST` and `:IFTEST` blocks of functions.
- The `:EX` statement provides a place to compute something before taking the next decision in cascading decision statements (such as:AND/:OR extensions of :IF/:WHILE).
- The `⎕EM` system function returns the Error Message part of `⎕DM` (the first line up to but not including the first `⎕TCNUL`).
- The `⎕DM` system variable supports localization and assignment.
- `⎕AT` and `⎕SIZE` allow scope left arguments.
- Orphans are automatically removed from the workspace on all `)LOAD`, `)COPY` and `)SAVE` operations.
- The [Config]VirtualCommitPrompt INI parameter controls what happens when virtual memory cannot be committed.
- The keyboard shortcut for the Edit/Replace dialog was changed to Ctrl+H. This makes APL+Win consistent with standard Windows Find/Replace key bindings and avoids the undesirable behavior of Ctrl+H destroying the selection rather than bringing up the Replace dialog.

Why These Features?

Most features are aimed at providing tools to help developers build *robust* applications *rapidly*. Developing applications rapidly is useless if they are unreliable. However, we have never given developers tools to support programming and testing practices that help them build robustness into their applications.

The goal of the debugging and testing features is to make it almost as easy to pay attention to robustness as to forget about it. Validation logic that executes only in debug mode can be used liberally without worrying about its impact on performance, since it can be disabled in the released system.

Because validation code doesn't need to be removed to disable it and can be reactivated in the field, developers can be more confident than ever in their ability to quickly diagnose and fix problems in deployed applications.

The new testing tools are intended to support cyclic test-while-coding mythologies such as Extreme Programming[1] and other kinds of Agile Software Development. APL's flexibility in adapting to new problems also increases the risk of breaking existing code. Testing to make sure code works correctly in a new mode does not ensure the changes haven't broken it for use in some old mode. Unit testing helps ensure the system as a whole and each of its components individually continue to fulfill the obligations they were designed for, especially the subtle ones we may have forgotten about. It is a much better approach to robustness than just "running the application for a while" after changing it.

The crash logging and MiniDump features were added to enable APL2000 to remotely diagnose and fix system bugs more quickly than ever before. The execution trace logging and detailed SI state information included in the log files should also help application developers to diagnose and fix their bugs as well.

The application restart features and improvements to the error messages boxes seen by users allow applications to behave more professionally and responsibly in the event of system failures. Restarted application can offer the option to the user (or do it automatically if pre-authorized) to upload log and MiniDump files to the application vendor for analysis and these can be forward to APL2000 for crash analysis.

The remaining features such as inline control sequences and other flow control structure changes, enhanced and simplified error handling, Unicode clipboard capabilities and filtered debugger stopping modes are intended to increase programmer productivity and joy. The next section discusses these features in detail.

New Features in Detail

Execution Speed Improvements

The execution speed improvements were achieved by critically examining all processes related to the main APL+Win interpreter 'loop' and modifying the applicable processes for maximum efficiency of operation. The benefits of the execution speed improvements in APL+Win v10.0 will generally be apparent without application system modification.

Some customers who have tested APL+Win v10.0 system report that their applications run in ½ the time they took on previous versions. These users didn't see as much an improvement in I/O intensive applications that spend time moving large amounts of data around on disk or across the internet. Calculation-intensive applications that "crunch" a lot of data in APL, as well as applications with extensive program logic, such as decision branching, iterative logic and control structures, generally run much faster in APL+Win v10.0.

[1] Extreme Programming: http://en.wikipedia.org/wiki/Extreme_programming

Enhanced Execution Speed Examples:

- Reduced interpreter overhead for control statements and tokens:

```
      ⍝ Version 10.0
      time
N: 20000000 Time: 4.992

      ⍝ Version 8.1.01
      time
N: 20000000 Time: 12.355

      ⍝ Version 3.5.01
      time
N: 20000000 Time: 6.833
```

```
    ∇ time;i;a;t;N
[1]   t←⎕ai[2]
[2]   N←20000000
[3]   :for i :in ⍳N
[4]     a←10+20
[5]   :end
[6]   ⎕←'N: ',(⍕N),' Time: ',⍕⎕ai[2]-t
    ∇
```

- Improved execution speed with data “crunching”:

```
      ⍝ Version 10.0                ⍝ Version 8.1.01              ⍝ Version 3.5.01
      calc 100                      calc 100                      calc 100
⎕dr:82  ⍴:50000 T:2.2         ⎕dr:82  ⍴:50000 T:4.711       ⎕dr:82  ⍴:50000 T:3.229
⎕dr:11  ⍴:50000 T:2.948       ⎕dr:11  ⍴:50000 T:5.897       ⎕dr:11  ⍴:50000 T:4.29
⎕dr:323 ⍴:50000 T:2.87        ⎕dr:323 ⍴:50000 T:5.538       ⎕dr:323 ⍴:50000 T:3.619
⎕dr:323 ⍴:50000 T:2.777       ⎕dr:323 ⍴:50000 T:5.491       ⎕dr:323 ⍴:50000 T:3.619
⎕dr:645 ⍴:50000 T:3.51        ⎕dr:645 ⍴:50000 T:6.833       ⎕dr:645 ⍴:50000 T:5.086
⎕dr:326 ⍴:5000 T:1.233        ⎕dr:326 ⍴:5000 T:2.449        ⎕dr:326 ⍴:5000 T:1.607
⎕dr:807 ⍴:5000 T:1.388        ⎕dr:807 ⍴:5000 T:2.808        ⎕dr:807 ⍴:5000 T:1.763
N:100 S:50000 Time:16.926     N:100 S:50000 Time:33.743     N:100 S:50000 Time:23.244
```

```
    ∇ S calc N;R;Z;I;T;L;R;Sdouble;Shalf;RR;LL;CC;C;NN;U;REP;CMP;Z2;Z3;S10;SS
[1]   :if 0=⎕nc'S'
[2]   :orif S=0
[3]     S←50000
[4]   :endif
[5]   :if N=0
[6]     N←10
[7]   :endif
[8]   NN←⍳N
[9]   S10←⌊S÷10
[10]  LL←(S⍴'A') (S⍴0) (S⍴2) (S⍴2) (S⍴1.1) (S10⍴(1 2) (3 4)) (S10⍴(1 2) 3)
[11]  RR←(S⍴'B') (S⍴1) (S⍴3) (S⍴4) (S⍴2.2) (S10⍴(5 6) (7 8)) (S10⍴6 (7 8))
[12]  CC←⍳⍴LL
[13]  T←⎕ai[2]
[14]  :for C :in CC
[15]    L←C⊃LL
[16]    R←C⊃RR
[17]    SS←⎕first⍴L
[18]    Sdouble←2×SS
[19]    Shalf←⌊SS÷2
[20]    CMP←SS⍴1 0
[21]    REP←SS⍴2 1
[22]    U←⎕ai[2]
[23]    :for I :in NN
[24]      :if C>1
[25]        Z←L+R
[26]        Z←L-R
[27]        Z←L×R
[28]        Z←L÷R
[29]        Z←L⌈R
[30]        Z←L⌊R
[31]        Z←-R
[32]        Z←×R
[33]        Z←÷R
[34]        Z←⌈R
[35]        Z←⌊R
```

```
[36]        Z←⍎¨100⍴⊂'1+2-3×4÷5'
[37]      :endif
[38]      Z←Sdouble⍴R
[39]      Z←Shalf⍴R
[40]      Z←Sdouble↑R
[41]      Z←Shalf↑R
[42]      Z←Sdouble↓R
[43]      Z←Shalf↓R
[44]      Z←Shalf⌽R
[45]      Z←L,R
[46]      Z←L,[0.5]R
[47]      Z←L,[1.5]R
[48]      Z←⍉Z
[49]      Z←⌽Z
[50]      Z←⊖Z
[51]      Z←CMP\CMP/R
[52]      Z←CMP⍀CMP⌿L,[1.5]R
[53]      Z←L calcFoo R
[54]      Z←L calcFoo¨ R
[55]      (Z Z2 Z3)←R L Z
[56]    :endfor
[57]    U←⎕ai[2]-U
[58]    ⎕←'⎕dr:',(3↑⍕⎕dr L),' ⍴:',(⍕SS),' T:',⍕U
[59]  :endfor
[60]  T←⎕ai[2]-T
[61]  ⎕←'N:',(⍕N),' S:',(⍕S),' Time:',⍕T
    ∇


    ∇ Z←L calcFoo R
[1]   Z←R
    ∇
```

Application System Tuning To Select Workspace Size

The benefits of the increased maximum workspace size in APL+Win v10.0 may be maximized with some optional 'tuning' effort. Analyzing the workspace memory usage of an application system and adjusting the new workspace size settings will help to achieve optimal performance of the application system.

As workspace sizes increase it becomes more important to understand the huge impact workspace memory management can have on overall performance of an application. In most applications, the overhead of allocating, freeing, recycling, and reorganizing workspace memory can account for a greater fraction of time than is spent on actually "crunching" numbers. While this is all handled automatically "behind the curtains" for APL+Win applications, there are decisions developers can make and actions they can take that have a dramatic impact on this aspect of performance, either positively or negatively.

A great deal of workspace memory management tuning is handled automatically and adaptively by the APL+Win system as it "learns" how an application behaves and changes memory management strategies accordingly. But there are hints that the developer can give to the system that enable it to make fewer guesses and better decisions. Hence, it is possible for the astute developer to help improve performance if they are given the necessary tools and insights and are motivated to spend a little extra time to optimize their application with respect to workspace memory usage.

Performance tuning cannot be considered in a vacuum that looks only at how one instance of an APL+Win application performs in isolation. Balance is important and well done applications need to be considerate of other applications running at the same time on the same computer. Therefore, we will also be discussing how to be a good neighbor to other applications. Even from a selfish perspective, when running multiple instances of APL to work on different parts of a problem in parallel, being a good neighbor to your own application instances can improve the overall throughput of the system.

The 'interpreter tuning' (⎕it) function gives the application system programmer the ability to observe and precisely control APL+Win v10.0 memory usage in an application system. Using ⎕it "?" discloses the following options:

AuditRefcounts Audit object reference counts (System Failure for problems)

AuditRefcountsC	Audit object reference counts (cleans up problems)
AuditRefcountsS	Audit object reference counts (report problems without cleaning)
MaxAlloc	Returns two-element vector indicating the maximum size object that
SymbolTable	Returns a matrix of information about the symbol table
WsFrisk	Force immediate workspace frisk - detects inconsistent internal state
WsMerge	Force immediate workspace merge - partial freespace consolidation
WsPack	Force immediate workspace pack - complete freespace consolidation
WsResize	Force immediate workspace resize - commits memory
wsAllocsPerMerge	Controls how often memory is merged (allocs per merge)
wsAllocsPerPack	Controls how often memory is packed (allocs per pack)

wsAllocsPerResize	Controls how often memory is resized (allocs per resize)
wsResizeExcessFree	Memory resize factor (percent of free-space to commit in excess)
wsResizeExcessMax	Maximum excess memory commit size (bytes)
wsResizeExcessMin	Minimum excess memory commit size (bytes)
wsResizeExcessUsed	Memory resize factor (percent of used-space to commit in excess)
wsResizeMax	Maximum memory commit size (bytes)
wsResizeMin	Minimum memory commit size (bytes)
wsSplitsPerPack	Controls how often memory is packed (splits per pack)
wsStatistics	Returns a vector of workspace memory management statistics
wsUsage	Returns a matrix of workspace memory usage statistics

The syntax of ⎕it supports documentation, get and set operations:

- Documentation, e.g. ⎕it '?wsAllocsPerMerge' results in "Controls how often memory is merged (allocs per merge)
- Get, e.g. ⎕it 'wsAllocsPerMerge' results in an integer value
- Set, e.g. 'wsAllocsPerMerge' ⎕it 20000

To review current workspace memory usage:

```
      ⎕IT'?wsUsage'
Returns a matrix of workspace memory usage statistics.
It will contain a row for each memory slot that was
allocated at system startup (whether or not that slot has
any memory committed).  There will be 1 row for APL+Win
running on systems without 4-Gigabyte-Tuning (4GT) enabled
(http://msdn.microsoft.com/en-us/library/bb613473(VS.85).aspx)
and 1 or 2 rows for APL+Win with 4GT enabled.  The columns of
the result matrix are defined as follows:
[;1]  Bytes reserved (portion of WsSize)
[;2]  Bytes committed to APL objects
[;3]  Bytes currently in-use to store APL objects (Used)
[;4]  Bytes NOT currently in-use to store objects (Free)
[;5]  Bytes reserved for value stack
[;6]  Bytes committed to value stack
[;7]  Virtual address where this memory slot begins
NOTE: Referencing this result causes a WsPack to be done
      and can slow down performance if referenced too often.
NOTE: More columns may be added in the future.

      ⎕IT'wsUsage'
 1073414144 12288 4508 1073409604       0     0 2147483648
 1284833280     0    0 1283784672 1048576 4096  412024832
```

Based on this ⎕it information, the ⎕it options as well as the APL+Win start-up parameter WsSize or the new start-up parameters WsSizeLow and WsSizeHigh, WsStackMin, WsStackMax, can be appropriately set to achieve maximum performance. The new start-up workspace parameters are specified as follows:

APLW.exe ... WsSize ...
APLW.exe ... WsSize WsMinLow ...
APLW.exe ... WsSize WsMinLow WsNotLow ...
APLW.exe ... WsSize WsMinLow WsNotLow WsNotHigh ...

These parameters are positional in that their order relative to each other matters. But they do not need to come at the beginning of the command line. That can come at the start, end, or intermixed with other parameters. For example, you can specify something like this:

APLW.exe 500M MyApp.INI 200M MyWorkspace.w3 100M

These size parameters can also be specified by name in the INI file (as part of the [Config] section such as [Config]WsSize) or on the command line with a [Config] prefix such as:

APLW.exe ... [Config]WsMinLow=500M ...

In this case, since [Config] prefixed parameters are not positional, they can appear in any order on the command line.

The WsSize parameter specifies the overall workspace size requested. The WsMinLow specifies the minimum amount of that WsSize that should be allocated in low memory. By default, memory is preferentially allocated in high memory, if available, before allocating any low memory. Use the WsMinLow parameter to force a minimum amount of memory to be allocated in low memory.

The WsNotLow and WsNotHigh parameters specify how much memory to not allocate in low and high memory. This is the amount of non-workspace memory to be reserved in both regions. You can also control SI stack allocation limits via the [Config] WsStackMin and [Config]WsStackMax parameters. These are not positional parameters on the command line, but as with workspace size parameter, they can be specified on the command line using the [Config] prefix such as:

APLW.exe ... [Config]WsStackMin=10M ...

Sizes may be specified using integer or decimal notation including a denominational suffix (listed below). For example, you can request a 1.5 Gigabyte workspace with at least 200 Megabytes allocated in low memory like this:

APLW.exe ... 1.5G 200M ...

Each memory size specification can be optionally followed by a denominational suffix from the following list (case insensitive):

K Kilobytes
M Megabytes
G Gigabytes
P percentage of physical memory (up to 2 GB - this is legacy)
% percentage of available virtual memory

The P notation (percent of physical memory) is a legacy parameter retained for compatibility with previous versions. It specifies the percent of total physical memory on the machine, up to the first 2 GB. On machines with more than 2 GB of memory, this parameter still is limited to 2 GB maximum. The % notation is a percent of available virtual memory. For the WsSize parameter, this is the percent of Total Virtual Memory available in both low and high memory. For WsMinLow and WsNotLow, this is the percent of virtual memory available in low memory (below the 2 GB line). For WsNotHigh this is the percent of virtual memory available in high memory (above the 2 GB line). For WsStackin/WsStackMax this is scaled relative to the WsSize setting.

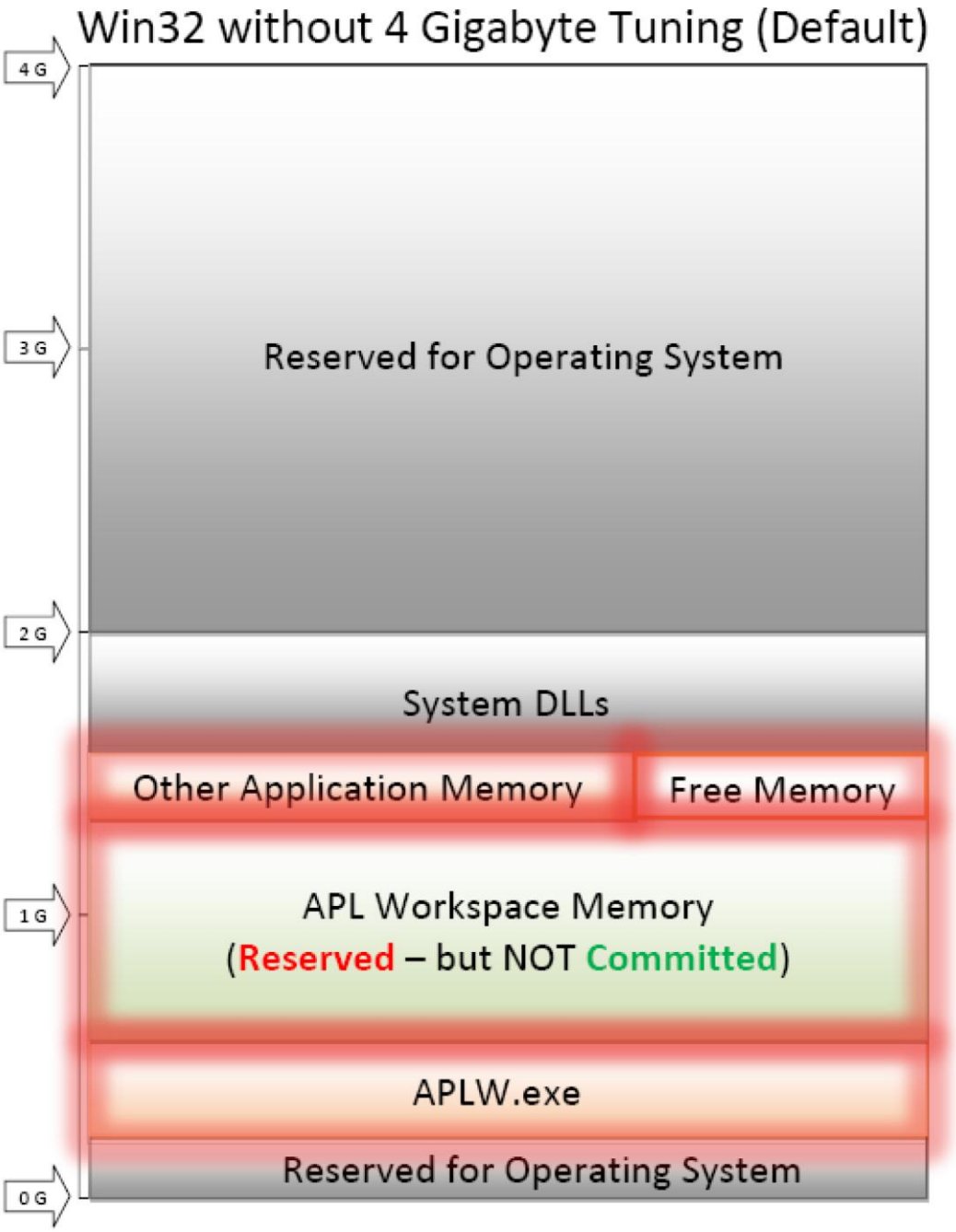
Insider's Tour of APL+Win Memory Architecture

The enhancement of maximum workspace size was implemented by keeping APL+Win a 32-bit application system so that most existing ActiveX and DLL components, as well as assembly language programs and other `⎕call` code, will continue to work without modification. Part of the effort in implementing this enhancement lays the groundwork for a possible future 64-bit version of APL+Win.

On a 32-bit Windows operating system version, a 32-bit application is normally limited to the address space below the 2GB address pointer. The '4GB tuning option' allows 32-bit application address up to 3GB on a 32-bit Windows operating system.

On a 64-bit Windows operating system, a 32-bit application system (WOW65) is limited to the address space below the 4GB address pointer.

Windows operating system dlls are immovably-positioned in a manner which fragments the memory space available to application systems into two large, non-contiguous blocks. Previously APL+Win utilized only contiguous memory, but APL+Win v10.0 was enhanced to support non-contiguous memory.

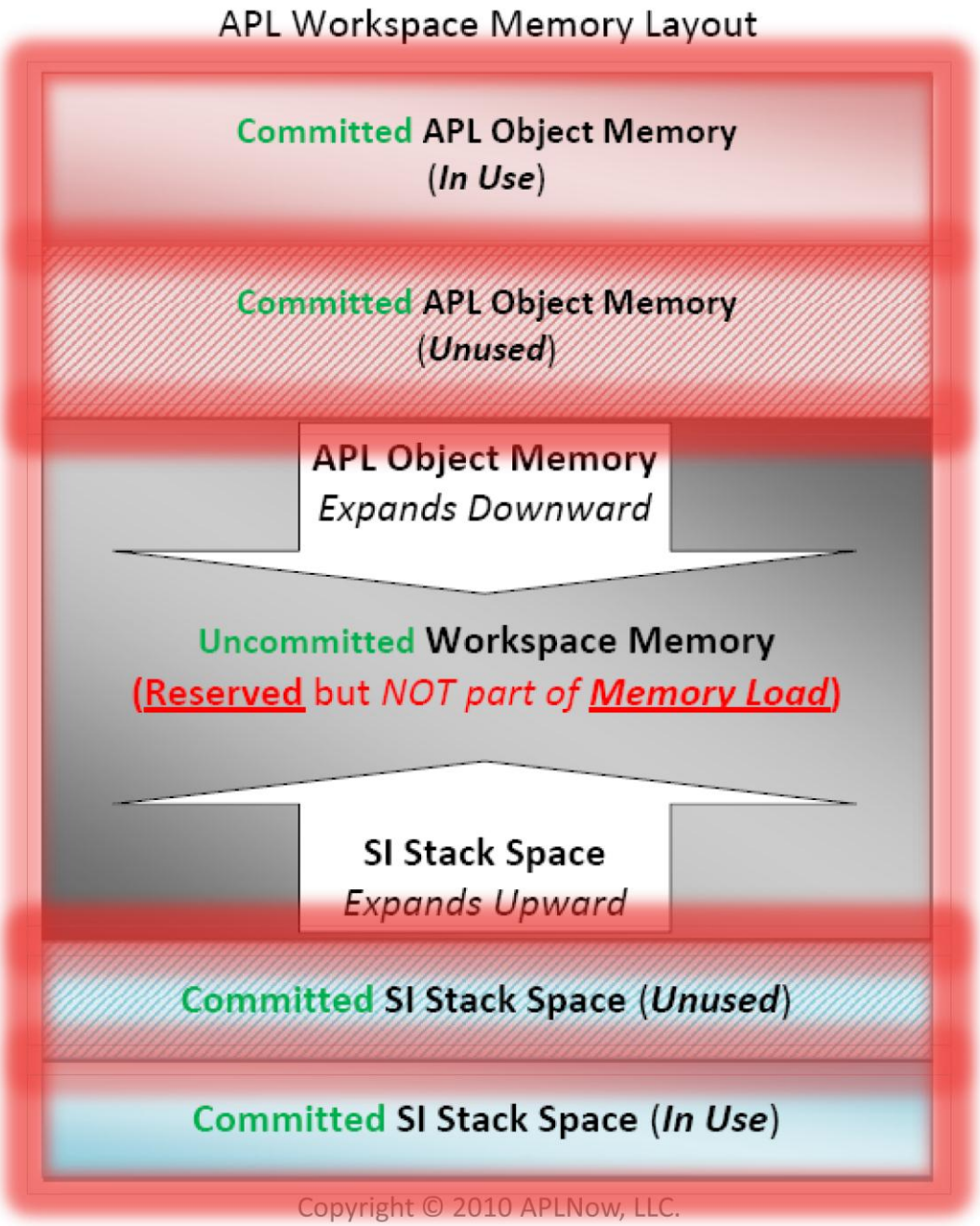


The WsSize start-up parameter controls the maximum workspace size for the application system. The default workspace size is 75% of the first 2GB of physical memory. Memory is reserved, but not committed, in one contiguous block at the start of execution and not released until `)off`.

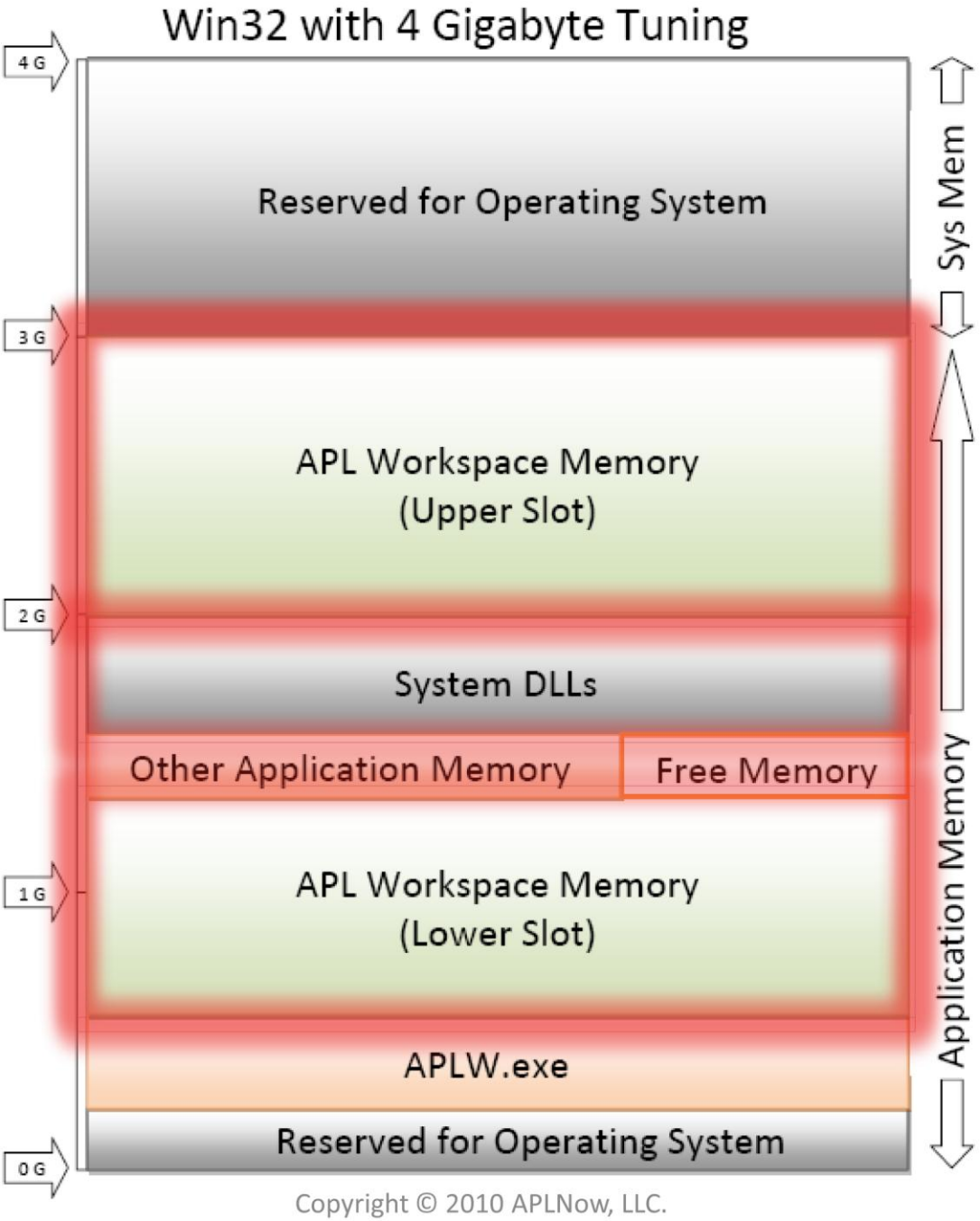
The remaining memory available to the application system is used by `⎕WI`, `⎕WCALL`, dlls, ActiveX objects and the APL+Win session manager.

APL+Win workspace memory, which remains constant, is divided into three regions:

- Committed APL+Win object memory (arrays, symbol table, etc.) The object memory is divided into in-use and un-used regions
- Committed ⎕SI Stack Space (execution state, values of locals)
- Uncommitted memory available for either APL objects or stack space



On Windows 32-bit operating system with '4GT tuning' on, more memory can be allocated to an application system. The larger available workspace size is not available as a contiguous block, so the APL+Win memory manangement methodology was enhanced to support two memory blocks separated by the immovable operating system dlls.



There is ‘no free lunch’ however:

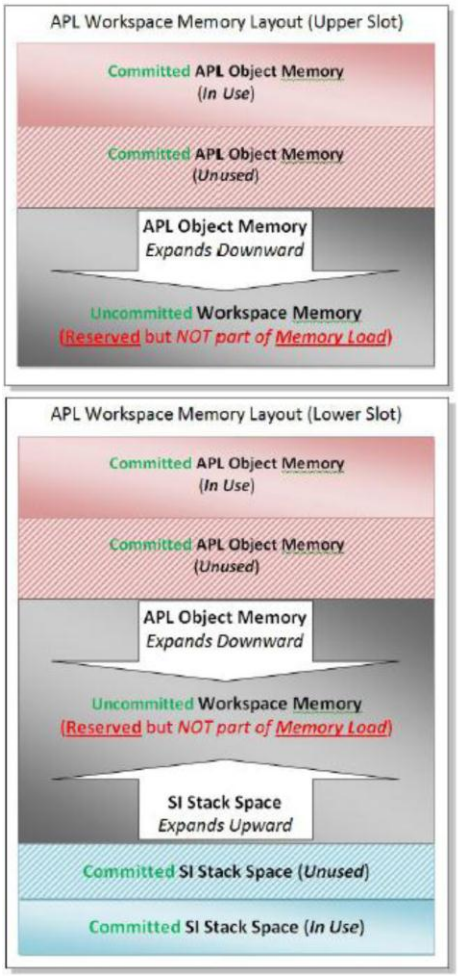
- The ‘4GT tuning’ option reduces the size of the file cache, paged pool and non-paged pool which can adversely affect application systems with significant networking or file I/O
- The ‘4GT tuning’ option must be explicitly enabled
- The hardware must have more than 4GB of RAM
- The increased workspace size may ‘starve’ other simultaneously-running application systems for memory.
- Some ActiveX objects and dlls may fail if consigned to a memory address above 2GB.
- Some pre-loaded dlls can limit available workspace size.

‘Segmented’ memory allocation for the APL+Win-based application system with ‘4GT’ On:

- The start-up WsSize parameter controls overall WS size
- Default WsSize is 75% of available virtual application memory
    - On a 32-bit OS, this is approximately 2.25GB
    - On a 64-bit OS, this is approximately 3.00GB
- The upper memory segment is all above the 2GB address
- The lower memory segment uses the remaining WsSize below the 2GB address
- If used, the minimum lower memory segment size is 128Kb
- If the requested WsSize completely fits in the upper memory segment, no lower memory segment is used
- The new start-up parameters, WsSizeLow and WsSizeHigh may be used to explicitly control the size of the upper and lower memory segments
- The default memory size or WsSize or both WsSizeLow and WsSizeHigh may be used
- Negative parameter values specify the amount not used
- Parameters can specify units by suffixes:
    - % Percent of Virtual Memory
    - P Percent of Physical Memory
    - G Gigabytes of Memory
    - M Megabytes of Memory

- K Kilobytes of Memory
- Bytes specified with no suffix

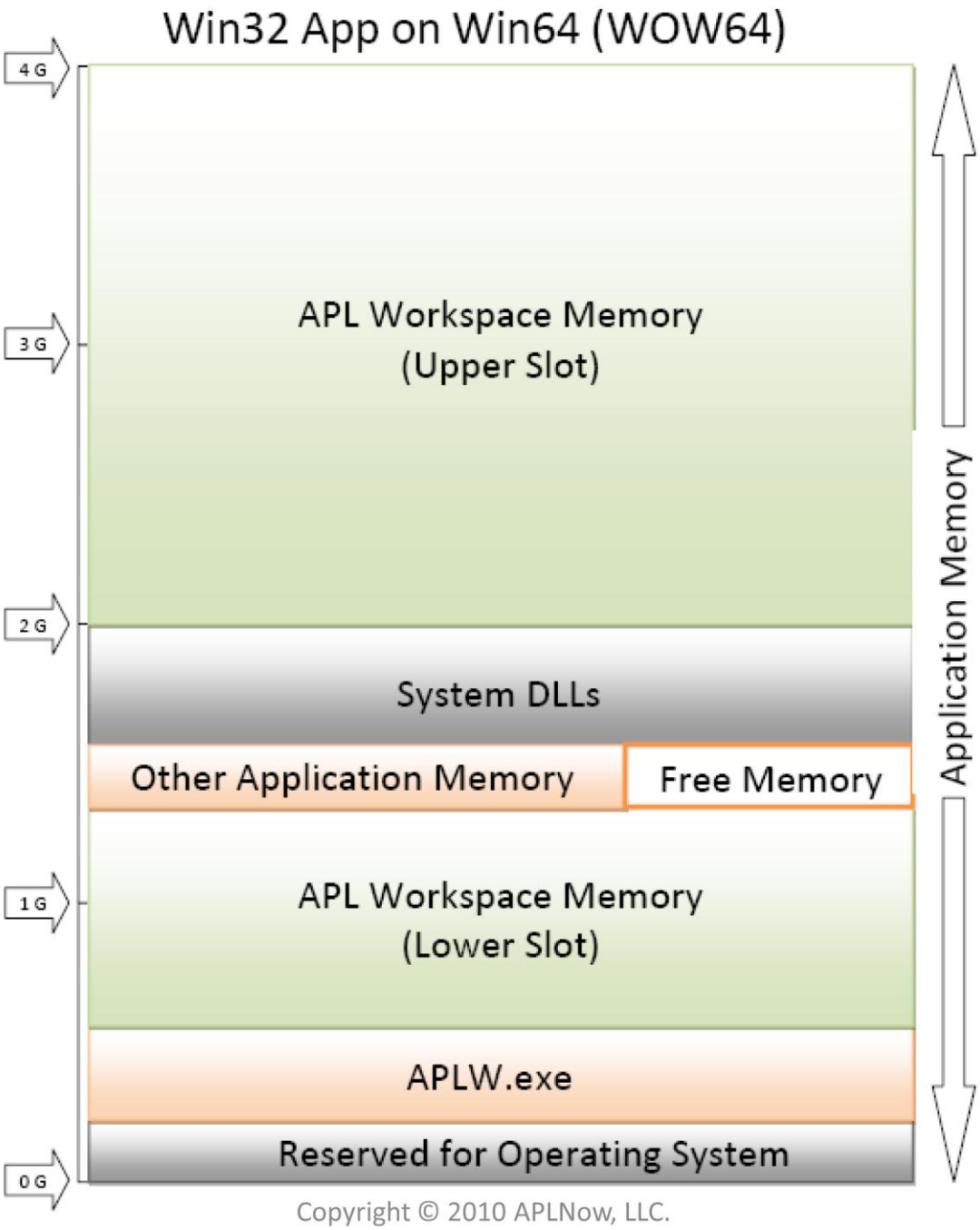
Allocation of Workspace Memory with '4GT' On:



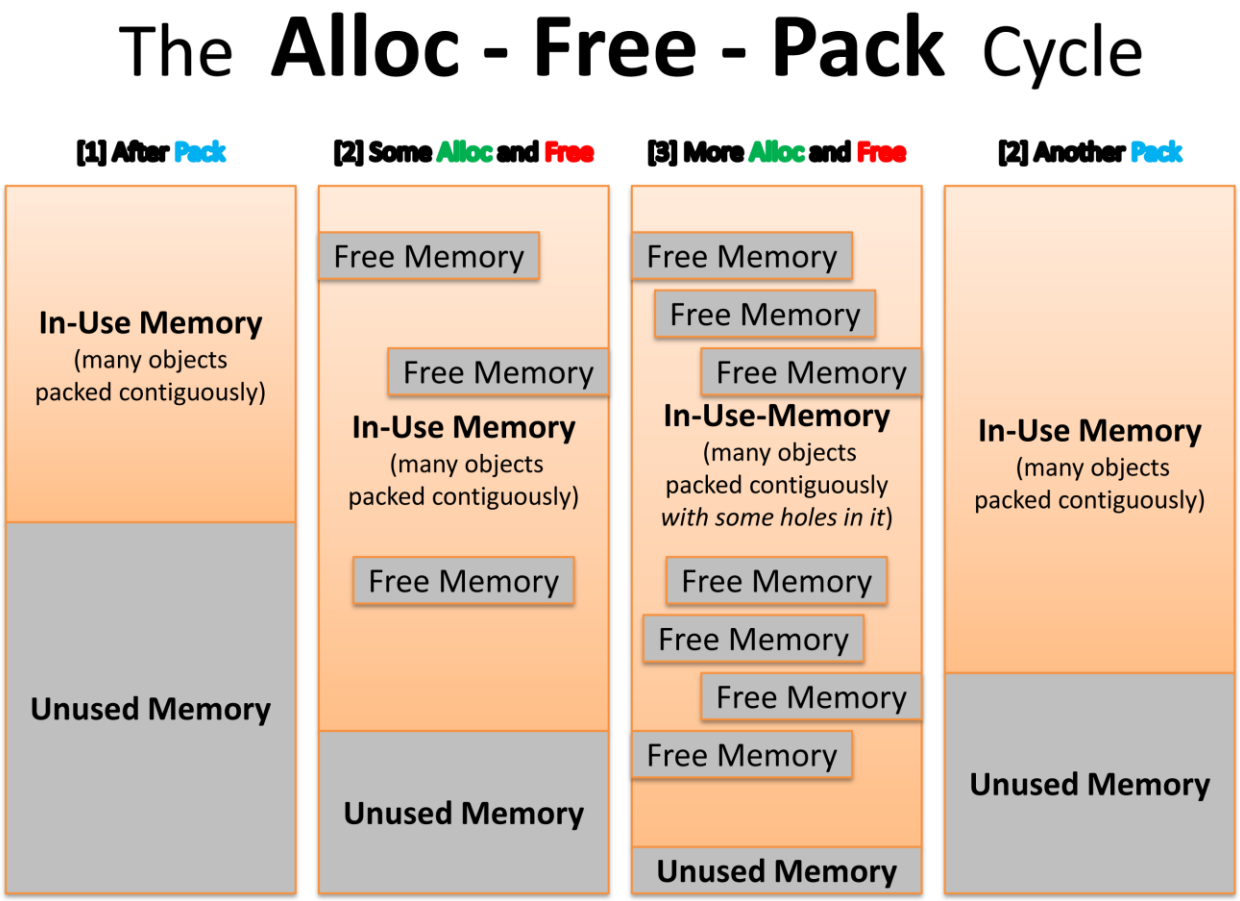
- **APL Objects** allocated in **Upper** slot first; then in **Lower** slot (only after **Upper** slot is too full for next alloc)
- **Unused Memory** may exist in both **Upper** and **Lower** slots concurrently
- **APL Objects** cannot span across slots (must completely fit in one slot)
- Might not be possible to allocate very large APL objects even when enough □WA appears to be available (because workspace is not one contiguous slot and objects cannot span across slots)
- **SI Stack** always allocated in **Lower** slot (never expands into **Upper** slot)

When APL+Win is used on a 64-bit Windows operating system:

- ‘4GT’ is automatically on
- Available application system memory is increased from 3GB to 4GB
- APL+Win runs in 32-bit compatibility mode (WOW64)



During the operation of an APL+Win application system, the workspace memory undergoes periodic reorganization as follows:



This diagram is simplified and does not illustrate the following additional functions which may apply when the APL+Win workspace memory is reorganized during application system execution:

- Freed objects are **Stored on a Free List** (a set of several link lists aggregating objects of similar size ranges together)
- New allocations **Search the Free List** before allocating objects from Unused Memory –If necessary, objects from the Free List are **Split** into used and free parts (if they are larger than the requested size) and the free part is put back onto the Free List
- Free List entries are periodically **Merged** to consolidate contiguous free objects into larger blocks. The result of merging is a transformed Free List containing larger blocks
- If there isn't a sufficiently large object on Free List, we **Allocate Unused Memory** for the object
- If there isn't sufficient Unused Memory for the allocation we **Resize** the workspace by adjusting the Committed size to increase the size of Unused Memory
- A **WS FULL** error is signaled if we cannot Commit a sufficiently large block of Unused Memory

What is the largest allocable APL+Win object size?

- When only one memory segment is employed by the application system, ⎕wa may be used.
- When '4GT tuning' on a 32-bit operating system or a 64-bit operating system is used and two memory segments are employed, ⎕wa cannot be used because it reports total memory available, but an APL+Win object cannot span two memory segments. In this case ⎕it 'MaxAlloc' which returns a 2-element vector:
    - Element #1 is the maximum size object that can be allocated in the current workspace memory
    - Element #2 is the remaining workspace size after allocating the maximum size object\

```
      ⎕WA    ⍝ Only reliable in ONE-SLOT Workspace scenario
2366041684
      ⎕IT 'wsUsage'
 1073414144 12288 4508 1073409604       0     0 2147483648
 1292632064     0    0 1291583456 1048576 4096  521076736

      ⎕IT'?MaxAlloc'
Returns two-element vector indicating the maximum size object that
can be allocated and the available space remaining after allocation.
If only one memory slot is available the second element will be zero.

      ⎕IT 'MaxAlloc'
1291583456 1073409604
```

Remaining size after allocating maximum size object

Maximum size object that can be allocated in current WS

Debug and Release Execution Modes

The *)DEBUG* command can query or change the debug/release execution mode. This can also be done programmatically via *⎕SYS*. Debug mode is a property of the active workspace. It is saved and loaded with it (*ON* by default). The *DebugOverride* configuration parameter can temporarily reinstate debug mode in a release mode workspace. This can be useful for debugging deployed applications.

When debug mode is enabled the *:DEBUG*, *:TRACE*, *:ASSERT*, and *:IFDEBUG* statements behave as described below. Whenever debug mode changes, all functions in the workspace are "reflowed". In release mode this causes debug statements to become logically invisible as if they were comments. Reflowing calculates the next-statement pointer for normal statements to make debug statements completely skipped without incurring any execution overhead, not even one machine cycle. In release mode, debug statements are not just inexpensive, they are literally *free*. *⎕IT*'s Internal Representation (*IR*) command lets us examine this in detail. When debug mode is *OFF* we have the following:

```
)debug off
'IR' ⎕IT 'Foo'
```

```
... Next/False Stmt [Line] Code
...    4          0 [0]    Foo
...    4          1 [1]    :debug ⎕←'debug'
...    4          2 [2]    :trace ⎕←'trace'
...    +          3 [3]    :assert 0<⍴⎕←'assert'
...    8          4 [4]    ⎕←'normal'
...    +/8        5 [5]    :ifdebug
...    8          6 [6]      ⎕←'ifdebug'
...    +          7 [7]    :end
```

Some output from these commands is omitted for brevity. The *Next/False* column shows the next statement to execute following each statement. The */False* part only exists for "decision" lines (such as line [5]), which must make true/false decisions about what statements come next. The next statement for line [0] tells us the first line to execute. This is set to 4 meaning the first three lines of debug statements will be completely skipped and execution will start on line [4] with the "normal" line. Its next statement is 8. This skips over the remaining debug statement and exits from the function. When debug mode is *ON*, this flow pattern changes completely as shown below:

```
      )debug on
      'IR' ⎕IT 'Foo'
... Next/False Stmt [Line] Code
...    +          0 [0]    Foo
...    +          1 [1]    :debug ⎕←'debug'
...    +          2 [2]    :trace ⎕←'trace'
...    +          3 [3]    :assert 0<⍴⎕←'assert'
...    6          4 [4]    ⎕←'normal'
...    +/8        5 [5]    :ifdebug
...    8          6 [6]      ⎕←'ifdebug'
...    +          7 [7]    :end
```

A plus (+) is displayed in the *Next/False* column when the next statement immediately follows the previous one. The (+) shown as the next statement for line [0] means execution begins at line [1] with the first debug statement. This is followed by two more debug statements until we reach the "normal" line and then continue with the final debug statement.

Even when debug mode is *ON* the flow optimizer avoids executing unnecessary lines (for example, lines [5] and [7] are never executed). When line [4] finishes execution we continue on line [6] without executing line [5]. Logically we need to execute line [5] to make a debug or non-debug decision about whether line [6] should to be executed or not. However, when we reflow the function for debug mode, we already know the decision on line [5] will always be true so we can skip that line and jump directly to line [6] for any line that would normally flow into line [5]. Similarly, when line [6] finishes execution it skips over line [7] because the flow optimizer knows there is nothing to execute there.

Execution of this code with debug mode *OFF* and *ON* illustrates these modes in action:

```
      )debug off
      Foo
normal
      )debug on
      Foo
debug
trace
assert
normal
ifdebug
```

Testing

Testing code can be defined inside :*TEST* … :*ENDTEST* blocks. A test block contains an initialization section followed by any number of :*PASS* or :*FAIL* clauses as illustrated by the example below:

```
      ∇ Z←A times B
```

```
[1]   Z←A×B
[2]   :test
[3]     A←10
[4]     B←20
[5]   :pass
[6]     :assert 200≡A times B
[7]   :fail
[8]     :assert A≡B
[9]   :pass
[10]    :assert B≡A+10
[11]  :fail 'INDEX ERROR'
[12]    (⍳10)[B]
[13]  :endtest
    ∇
```

During normal execution, the `:TEST` block in `times` is logically invisible. Line `[1]` executes and returns without executing the test code. But when `times` is selected by the argument[2] of `)TEST` or `⎕TEST` line `[1]` is skipped and only the test code is executed. One test can invoke others via the `⎕TEST` function.

As with normal function execution the ambient state of system variables such as `⎕IO`, `⎕CT`, etc. is used during testing. Test code can set these variables if it needs to assure they are in a given state.

The code in test blocks *always* executes in debug mode. Therefore, the `:ASSERT` and other debug statements always execute in that context. However, functions called by the test code executes in the workspace's debug/release mode.

When test execution begins for the example above, the initialization section sets `A` to `10` and `B` to `20`. Then the tests defined by `:PASS` and `:FAIL` clauses are executed in sequence. Any tests that fail are noted in the test log and execution continues with the next `:PASS` or `:FAIL` clause.

A future version may support declaration of local variables specific to the each `:TEST` block but this version runs tests using the function's local variables. Variables set in one test flow into the next. If a function contains multiple `:TEST` blocks they each execute independently as a new invocation of the testing context. Local variables set in one `:TEST` block do flow into subsequent `:TEST` blocks when present in the same function but not when in different functions. However, global variables, the state `⎕WI` objects, files ties, external object, etc. *do* flow between one `:TEST` block and another and therefore execution order can be important.

`:PASS` clauses are expected to run without unhandled errors. A `:PASS` clause may contain setup calculations and one or more `:ASSERT` statements to check that results match expectations. If an error occurs the test will be considered to have failed. The remaining lines in the `:PASS` clause will be skipped and execution will continue with the next `:PASS` or `:FAIL` clause. The use of `:ASSERT` is *essential* in tests! If line `[6]` didn't include an `:ASSERT`, an inequality of the `200≡A times B` test would *not* be detected as a failure because no error would have been signaled.

`:FAIL` clauses are expected to cause an error. The test is considered a success if the expected error occurs and a failure if the error does not occur. Errors are only expected to occur on the last statement of the `:FAIL` clause. Errors occurring before the last statement are considered to be setup failures and cause the test to be considered a failure. An optional filter argument can denote a specific error using the same syntax as `:CATCH`. Any error is ok if a filter is not used. An `INDEX ERROR` is anticipated on line `[12]` because of the filter specified on line `[11]`. Other errors are not accepted.

The `:IFTEST` … `:ENDIFTEST` block defines code that should only run while testing. It is not the same as a `:TEST` block. When testing starts the workspace is reflowed to make `:IFTEST` blocks logically visible. They behave like comments when testing is not being done. `:IFTEST` blocks are useful during development when bridging code is needed to fill in for parts of the application that are not yet functional. This allows the testing cycle to begin earlier during development than would otherwise be convenient. Using `:IFTEST` for bridging code is superior to placing it directly in the function because it makes its

[2] The `)TEST` and `⎕TEST` argument uses a wildcard notation to select functions for testing. For example, `⎕TEST "Foo ⋆_test"` selects `Foo` and all functions whose names end with `_test` for testing. Of course, only those functions selected by the argument that contain test blocks will actually be tested).

purpose more visible, its effects transient, and helps avoid forgetting to remove it when no longer needed. Any `:IFTEST` blocks executed during testing are noted in the test log as a reminder to remove them later.

The TraceLog Feature

The trace log is a hyper fast in-memory wrap-around buffer that can capture a history of over 134 million of the most recently executed statements with very low overhead. It can be useful for exploring how program logic goes awry without modifying the code or changing executing timing enough to make bug mutation likely. Triggers can be defined to dump the trace log to file whenever an error filter pattern is matched and/or at program exit. Trace logging is controlled via four configuration file parameters:

`TraceLog`=n	Initial trace buffer size
`TraceLogAlloc`=n	Maximum trace buffer size
`TraceLogExit`=1 or 0	Controls whether trace buffer is dumped to file upon exit or not
`ErrorLogFilter`=filter	Error filter pattern that triggers dumping of trace buffer to file

and four `⎕IT` commands:

`log←⎕IT 'TraceLog'`	Return the current trace buffer content as a character vector
`'TraceClear'⎕IT size`	Clear and/or resize the trace buffer (up to `TraceLogAlloc` max)
`'TraceMark'⎕IT mark`	Insert an identifying "mark" into buffer (useful during analysis)
`'Dump'⎕IT label`	Dump the trace buffer (and other internal state info) to file *now*!

Refer to the separate document, "APL+Win v10 – TraceLog Execution History.PDF" for more information.

Crash Diagnoses and Recovery

If a crash occurs, the system now creates a log file containing extensive contextual information including the reason and location of the crash, the `TraceLog` execution history leading up to the crash, the SI state with partial[3] values of all local variables at each stack level at the time of the crash, along with user selectable global variables, and a MiniDump file. The MiniDump file can be executed by APL2000 with a post-mortem debugger to either help diagnose and fix bugs in the APL+Win system or determine if the crash is happening in a third party component. In that case, the third party may be able to use the MiniDump file to diagnose their bug. The system also writes an Error Event about the crash in the Windows Event Log. All debugging files can be encrypted if an application chooses to make them secure.

In addition, a new crash recovery mechanism allows applications to either restart themselves or launch another application in response to a system crash. This can be done as simply as entering a single line in the configuration file like this:

```
RestartCmd=""%EXEPATH%" "%INIPATH%" "%WSDIR%recover.w3""
```

Many possible substitution `%variables%` may be used in addition to those shown in the example above.

Whenever the recovery application is started, the system adds a parameter to the command line giving the name of the crash log file. The application can use this as a signal it is being restarted in response to a crash and display a crash recovery dialog - rather than its normal startup dialogs - informing the user about the crash and asking what action should be taken next (such as upload the log and MiniDump files to the vendor's servers for analysis or restart using the application).

If the restart feature is not used, the system displays an improved message box containing vendor contact information from parameters in the configuration file, which refers them to the log file for more

[3] A "value tip" containing up to 1024 characters (configurable length) of data from the beginning of each local variable's present value and localized previous value are displayed in the extended SIE listing along with other internal state information including the source code line and internal PCODE representation of that line for each SI level.

information. The text of the message box is completely configurable and parameters are also available to suppress message boxes for cases where the application is running in a server context.

Refer to the separate documents, "APL+Win v10 – Crash Log File Encryption & Decryption Mechanism.PDF" and "APL+Win v10 - Crash Recovery.PDF" for more information.

Inline Control Sequences

NOTE 1: The ICS feature sets has been made experimental in version 10.0. It is only enabled for execution in the development system and not allowed in the runtime system. Enabling for the development system requires the following entry in the INI file:

[Experimental]
EnableICS=1

If ICS features are executed without this feature being enabled (in runtime system regardless of INI file setting or in development system without EnableICS=1) then the following error message is given:

```
SYNTAX ERROR: Experimental ICS features not enabled
```

No message box prompts are displayed in either development or runtime system. The only failure indication is the error message above. The value must be in the INI file at system startup time. It is read from the INI file only once. Subsequent changes, after APL has started have no effect on the behavior until the next time APL is started.

NOTE 2: The :RES clause of the :RETURNIF statement is implemented internally as an ICS expression. This means that the :RES clause is also experimental for version 10.0. This is not a philosophical decision. Just a practical one in order to avoid having to modify the way the :RES clause is implemented. This does not affect the :RETURN statement with argument. It is not tied to the EnableICS setting and is not experimental.

Inline control sequences (ICS) can articulate arbitrarily complex logical calculations with a concise and efficient notation, which uses colon-prefixed keywords `:AND`, `:OR`, `:THEN`, `:ELSE`, and `:CHOOSE` embedded in expressions. It uses Progressive Partial Evaluation (PPE) similar to the `&&`, `||`, and `?` operators found in C, C++, C#, Java, JavaScript, etc. ICS expressions are an inversion of traditional control structures. They may contain cascading decisions and execution alternatives in a single statement, whereas traditional control structures involve multiple expressions spread across multiple statements typically on multiple lines. PPE is nothing new to APL. The `:IF` control statement uses PPE to jump into the True or False clause as soon as the overall outcome can be deduced from partial results. For example:

```
:IF Test 1
:ANDIF Test 2
:ANDIF Test 3
    True
:ELSE
    False
:ENDIF
```

If any of the test conditions are false the remaining tests are skipped and the False clause is executed immediately. If a test condition is true, the next test must be evaluated to determine the outcome of the calculation. The True clause is only executed if all tests are true. Step by step we have: if Test 1 is false, Test 2 and Test 3 are skipped and the False clause is executed. If Test 1 is true, Test 2 is evaluated. If it is

false, Test 3 is skipped and the False clause is executed. If true, Test 3 is evaluated. The True clause is executed only if Test 3 is also true. Otherwise the False clause is executed.

Similarly, when the `:IF` statement is followed by a series of `:ORIF` tests, we skip the remaining tests and begin execution of the True clause as soon as *any* of the test conditions are True. We only continue evaluation of tests and eventually execute the False clause if all previous test conditions are false.

PPE is a powerful tool for simplifying program logic. For example, the first test in a series might determine if a function's left argument has a value (i.e., whether called monadically or dyadically) and a second test might check its rank or shape. If the argument doesn't have a value we discontinue the test sequence early since we cannot check its rank or shape without a value. By guarding later tests with previous tests that pre-screen certain conditions (such as existence of a value) we can efficiently and concisely test the conditions we want with a minimum of coding and a maximum of clarity.

The `:IF … :ANDIF` logical progression shown above using five statements can be stated more concisely using one ICS expression like this:

```
(Test 1 :AND Test 2 :AND Test 3 :THEN True :ELSE False)
```

The order of progressive evaluation above might not feel right to everyone at first glance. APL executes from right to left and this might feel like it's going the wrong way. While there isn't enough space in this paper to fully discuss the decisions that went into ICS design the following summary is offered:

- APL executes from right to left but is conceptualized, written, and read from left to right[4]. We usually conceive of and type the left part before the right and always read it that way.
- Conditional expressions are usually typed with the condition to the left of the objective:

  ```
  →(Condition)/L1                    ⍝ for branching
  ⍎(Condition)/'Expression'          ⍝ for execute
  ⎕ERROR(Condition)/'ERROR MESSAGE'  ⍝ for signaling errors
  ```

  This is a good thing because we tend to think of the condition before we think of the objective, or at least we think about how to express the condition before the objective. Therefore, this ordering minimizes the number of superficial left and right key movements required to type the expression.
- Progressive logical decisions are generally conceptualized in their order of dependency. For instance, we need to know if a left argument exists before we test its rank or shape. So we tend to think of existence testing before rank or shape testing.
- The left to right order of ICS progressions is reflective of this natural thought order.
- If ICS expressions were evaluated right to left like this:

  ```
  (True :ELSE False :WHEN Test 3 :AND Test 2 :AND Test 1)
  ```

  we would have to type the tests in the reverse order from how we would naturally think of them.
- If we think of ICS expressions as being like diamonds with decisional powers then it is easier to understand why the left to right ordering makes sense. They follow the same flow pattern as diamond separated[5] statements and are consistent with a pattern of programming that was common in pre-control-structure days when hierarchical conditions would often be coded like this:

  ```
  →(Condition1)/⎕LC+1 ⋄ →(Condition2)/⎕LC+1 ⋄ ObjectStatement
  ```
- The keywords in an ICS expression are not functions or operators. They are conjunctive in nature and exist in a different semantic space than the functional sub-expressions connected by them.

[4] The statement that APL is "conceptualized, written, and read from left to right" is subject to debate but is consistent with the view held by every person the author has ever discussed this topic with.

[5] There probably still is not universal agreement that diamond separated statements are executed in the correct order. There are some who may still argue that they should not exist at all or that the rightmost statement should be executed before the left. However, the convention in all APL dialects known to the author executes them left to right.

- It can be useful to think of ICS keywords as being like parentheses with decisional powers. ICS keywords morph normal order of execution in a similar way and exist in a similar semantic space.

The following list shows all recognized forms of ICS expression (for brevity the *:OR* form of each expression is not shown; wherever *:AND* appears it can be replaced by *:OR*):

```
( Cond  :THEN   Case1 :ELSE Case0 )
( Cond  :THEN   Case1 )
( Cond  :ELSE   Case0 )
( Cond1 :AND    Cond2 )
( Cond1 :AND    Cond2 ... :AND  CondN )
( Cond1 :AND    Cond2 ... :THEN Case1 :ELSE Case0 )
( Cond1 :AND    Cond2 ... :THEN Case1 )
( Cond1 :AND    Cond2 ... :ELSE Case0 )
( Index :CHOOSE Case1 : Case2 : ... : CaseN :ELSE Else )
( Index :CHOOSE Case1 : Case2 : ... : CaseN )
```

Parentheses around ICS expressions are only necessary to the extent needed to denote logical groupings, alter progression evaluation order, or disambiguate cases where the colon-prefix of the first keyword would otherwise be interpreted as declaring the name to its left as a statement label.

Any statement beginning with a colon-space (or any other character that is not part of an identifier immediately following the colon) tells the interpreter to not look for labels in that statement. Either of the following statements may be used to prevent *Condition* from being interpreted as a label:

```
( Condition :THEN Action )
: Condition :THEN Action
```

The *:AND* and *:OR* keywords can be used in sequences of any length. However, they cannot be mixed in the same expression unless parentheses are used to group them like this:

```
( Cond1 :AND Cond2 ) :OR ( Cond3 :AND Cond4 :AND Cond5 )
```

The *:THEN* and *:ELSE* keywords offer a binary choice between two alternative expressions. Only one of the expressions is executed. These keyword can be used together or alone if only one choice is needed, with the implicit other choice being no execution and no result value. If both are coded *:THEN* must come before *:ELSE*. In contexts that require a result, both clauses are required.

The *:CHOOSE* keyword selects one expression from a list of choices separated by colons, which must be followed by a space or other character that doesn't begin a name. The value to the left of *:CHOOSE* must be a singleton integer. It is used as a *⎕IO* sensitive index to select one of the cases to execute. An optional *:ELSE* clause can be executed if there isn't a case corresponding to the index value. An error is not signaled if the index does not select a case unless the expression is used in a context that requires a value, in which case an *INDEX ERROR* is signaled.

The following *:CHOOSE* expression is roughly equivalent to the *:SELECT* statement below it:

```
( Index :CHOOSE CaseA : CaseB : CaseC :ELSE ElseCase )
```

```
:SELECT Index
  :CASE ⎕IO
    CaseA
  :CASE ⎕IO+1
    CaseB
  :CASE ⎕IO+2
    CaseC
  :ELSE
    ElseCase
:ENDSELECT
```

Except *:SELECT* cannot be used to produce an argument value in the way *:CHOOSE* can:

```
X Foo ( Index :CHOOSE CaseA : CaseB : CaseC :ELSE ElseCase )
```

The :LEAVEIF and :CONTINUEIF Statements

The `:LEAVEIF` and `:CONTINUEIF` statements provide conditional loop exit and continuation with a single statement. This would otherwise require the three equivalent statements shown on the right below:

```
:LEAVEIF I>N          ↔      :IF I>N ◊ :LEAVE    ◊ :ENDIF
:CONTINUEIF I>N       ↔      :IF I>N ◊ :CONTINUE ◊ :ENDIF
```

The :RETURN with Value and :RETURNIF Statements

The `:RETURN` statement can now directly return a result value via its argument. For example,

```
:RETURN X Foo Y Goo W
```

returns the expression's value in one statement rather than a pair of assign and return statements:

```
Z←X Foo Y Goo W
:RETURN
```

Conditional returns can be done via the `:RETURNIF` statement in either of the forms below:

```
:RETURNIF I>N
:RETURNIF I>N :RES X Foo Y Goo W
```

The `:RETURNIF` statement returns if its conditional argument is true or continues with the next statement if false. It is similar in concept to the `:LEAVEIF` statement and replaces three statements with one. In the case of `:RES` four statements are replaced. Whenever `:RETURN` has an argument or the `:RES` clause is used with `:RETURNIF`, the expression's value is implicitly assigned to the result variable before returning. The `:RES` clause is not executed unless the return is taken.

If the result expression returns no value, then the `:RETURN` statement or `:RES` clause also return no value. For example, if `Foo` returns no value then `:RETURN` will also return no value without an error:

```
:RETURN Foo X
```

In contrast, trying to assign a result variable like this `Z←Foo X` would cause a `VALUE ERROR`!

The ⎕VALENCE, ⎕MONADIC, ⎕DYADIC, and ⎕NOVALUE

`⎕VALENCE`, `⎕MONADIC`, and `⎕DYADIC` indicate how the current function was called. `⎕VALENCE` returns 0, 1, or 2 for niladic, monadic, or dyadic invocation or `¯1` if not a function (such as a callback). `⎕MONADIC` is equivalent to `(⎕VALENCE=1)` and `⎕DYADIC` is equivalent to `(⎕VALENCE=2)`.

`:RETURN` and `:RES` allow the NO-VALUE system variable `⎕NOVALUE` as an argument to explicitly indicate no value. Using `:RETURN` without an argument is not the same as `:RETURN ⎕NOVALUE` unless the result variable was valueless.

The :TRY Statement with ⎕ELX Localization

The traditional `:TRY` statement only handles errors in its immediate execution context. Errors in functions called by it are handled via the ambient `⎕ELX` handler in effect when the try block was entered. If the ambient handler pops unhandled errors back to lower `⎕SI` levels for handling the try block's `:CATCH⋆` handlers will get a chance to handle the error when it reaches their context. However, if `⎕ELX` has a value such as `'⎕DM'` execution will suspend at the point of error and not be handled by the try block.

An optional argument has been added to `:TRY` to allow a try-local `⎕ELX` handler to be specified. If invoked like this `:TRY '⎕THROW ⎕DM'` unhandled errors in called functions will be popped back to the try block for handing regardless of ambient `⎕ELX` setting. A star (⋆) used in the argument like this `:TRY ⋆` is a shorthand notation equivalent to this `:TRY '⎕THROW ⎕DM'`.

The :TRYALL Statement

The `:TRYALL` statement provides resume-on-next-line error handling, which in some cases is easier to use than `⎕ELX` or `:TRY ... :CATCH`. Errors occurring in its context are handled by skipping any remaining statements on the same line and resuming execution on the next line. This is similar to the effect of setting `⎕ELX←'→⎕LC+1'` but without the problems doing so could cause in called functions. Conditional statements like `:IF` and `:FOR` cannot be used in a `:TRYALL` block but unconditional statements like `:DEBUG` and ICS expressions can. A try-local `⎕ELX` argument or star (⋆) notation is also allowed.

The :CATCH and :LIKE Clauses and ⎕EM Variable

The `:CATCH` clause supports wildcard pattern matching to select errors in a `:TRY` block. It is more concise and efficient than using APL expressions in a `:CATCHIF` clause. For example, rather than this:

```
:CATCHIF 'DOMAIN ERROR'≡(∧\⎕DM≠⎕TCNL)/⎕DM
```

The same error can be selected with a `:CATCH` clause like this:

```
:CATCH 'DOMAIN ERROR'
```

Multiple errors can be separated by semicolon like this:

```
:CATCH 'DOMAIN ERROR;INDEX ERROR'
```

And wildcard characters can be used like this:

```
:CATCH 'FILE ⋆ ERROR'
```

The last example above matches `FILE INDEX ERROR`, `FILE TIE ERROR`, etc.

Patterns are matched with the new `⎕EM` value, which is the same as `(∧\⎕DM≠⎕TCNL)/⎕DM`. The control characters in patterns are: (⋆) matches zero or more of *any* characters; (?) matches one of any character; (@) is reserved for the future to denote a context such as a specific function; (⍴) is reserved for the future to denote a regular-expression; (;) separates patterns; (\) makes the control character following it behave as a literal (e.g., `"MATCH\⋆THIS"` matches `⎕EM` value `"MATCH⋆THIS"`).

Specifying a `:CATCH` clause without an argument is a synonym for `:CATCHALL` and handles any errors not handled by previous `:CATCH` and/or `:CATCHIF` clauses (if any).

The `:LIKE` clause uses the same pattern matching notation as `:CATCH` but appears as an alternative to the `:CASE` or `:CASELIST` clause in a `:SELECT` statement. Rather than matching the `:SELECT` argument by value equivalence, it selects character scalar and vector cases matching its pattern.

The :FINALLY Clause

The `:FINALLY` clause can appear as the last clause of a `:TRY` statement, following all `:CATCH`, `:CATCHIF`, and `:CATCHALL` clauses (if any). Some form of `:CATCH⋆` clause is normally required in a `:TRY` statement but when a `:FINALLY` clause is coded `:CATCH⋆` clauses are optional. The `:FINALLY` clause defines a block of code that always executes no matter how the `:TRY` statement is exited (whether by branch, return, error, continue, leave, etc). The only exceptions are actions such as `)LOAD`, `)RESET`, niladic branch (→), and system exit. Future versions may provide an option to finalize these exits too.

Code placed in a `:FINALLY` clause can be useful for cleaning up resources such as closing files and database connections but can also simplifies program logic in many other cases.

The :NEXTCASE Statement

The `:NEXTCASE` statement flows from one case of a `:SELECT` statement into the next without branching. This can be useful for parsing varying length arguments where each element gets special processed and then falls into the case for the next smaller length. For example, to handle up to 5 elements the cases can be arranged as `:CASE 5`, `:CASE 4`, `... :CASE 1` with the `:SELECT` statement called with the length of the argument array. If there are 5 elements `:CASE 5` will process its argument and then

`:NEXTCASE` into `:CASE 4` will in turn `:NEXTCASE` into `:CASE 3`, etc. When there are fewer elements execution will start somewhere below `:CASE 5` and follow the same pattern as described above.

Multi-Variable :FOR Statement

The `:FOR` statement now supports multiple variables strand notation like this:

```
:FOR a b c :IN (1 2 3) (4 5 6) (7 8 9) (10 11 12) (13 14 15)
```

In the first iteration `a b c` will be assigned `1 2 3`, in the second `4 5 6`, and so on.

The ⎕LOG Function

The `⎕LOG` function formats its right argument into a debug log file and/or the Windows Event Log. It may be called during debug or release mode. The log file is automatically created with a timestamp based filename and each entry in it is prefixed by a timestamp. Newline characters are normalized to the Windows CR+LF standard and APL characters are translated to ANSI to make the output viewable with any standard editor. If the `EncryptLog` option is enabled the log file is encrypted as it is written.

The optional left argument controls where the output is written. D selects the debug log file (default) while I, W, or E selects the Windows Event Log as Information, Warning, or Error events.

Refer to the separate document, "APL+Win v10 – Crash Log File Encryption & Decryption Mechanism.PDF" for more information.

The :DEBUG, :TRACE, and :IFDEBUG Statements

The `:DEBUG` and `:TRACE` statement's argument is executed only in debug mode. The `:TRACE` statement implicitly calls the `⎕LOG` function to write its argument's value to the debug log file but the argument expression is not required to return a value (if it does not, nothing is written the debug log file).

The `:IFDEBUG` statement defines a block of statements that execute in debug mode only. It must end with an `:ENDIFDEBUG`, `:ENDIF`, or `:END` statement. It can be coded with or without an `:ELSE` clause, which executes in release mode, with additional conditions, or a combination of both. For example:

```
:IFDEBUG
     Debug Statement(s)
:ELSE
     Release Statement(s)
:ENDIFDEBUG
```

```
:IFDEBUG Condition1 :OR Condition2
:AND Condition3 :OR Condition4
:AND Condition5
     Debug Statements(s)
:END
```

The :ASSERT and :VERIFY Statements

The `:ASSERT` and `:VERIFY` statements are identical except `:ASSERT` executes only in debug mode whereas `:VERIFY` is always executed. They evaluate their argument and signal an error if it returns a value and the value is not `1` or an empty character vector (`''`). The error message consists of the prefix `'ASSERTION FAILURE: '` followed by the text of the failing APL expression including comment or if the value is a character vector, the value of the vector. Typical usage patterns are:

```
:ASSERT (⍴X)≡(⍴Y) :AND (I>J :OR K<50) ⍝ Oops
:ASSERT X≡Y :ELSE 'X does not match Y'
```

If these assertions were to fail, the following error messages would result:

```
"ASSERTION FAILURE: (⍴X)≡(⍴Y) :AND (I>J :OR K<50) ⍝ Oops"
"ASSERTION FAILURE: X does not match Y"
```

New Unicode Clipboard Support in Session Manager

The “Enable Unicode Clipboard” item to the Edit menu (near where it lists Cut, Copy, Paste) when enabled (checked) allows Unicode text containing APL characters to paste into and from APL+Win. This is bound to the Ctrl+U key for toggling Unicode clipboard mode on/off. There is also a “UNI” indicator in the status bar when this option is enabled.

Change Edit/Replace Keystroke Shortcut to Ctrl+H to Session Manager

Added Ctrl+H as a keystroke recognized for doing Edit/Replace. This makes APL+Win consistent with standard Windows Find/Replace key bindings and avoids the nasty behavior we used to have of Ctrl+H being a backspace and destroying selection rather than bringing up the Replace dialog. The old Ctrl+R keystoke still works as before but no longer shows up on the menu (which now shows Ctrl+H).

Allow Runtime System to Load Development Workspaces

The runtime system (aplwr.exe) no longer restricts the kind of workspace it can load. Now it can load workspaces that were created via either the)SAVE and)RSAVE system commands without any differences in behavior except that runtime workspaces are encrypted. This enhancement only applies to runtime workspaces created in APL+Win v10.0.

The old rules still apply as far as the type of workspace the development system (aplw.exe) can load. If the workspace was saved with the)RSAVE system command, the runtime workspace may only be loaded in the development system that created the workspace.

⎕WGIVE Optimization

⎕WGIVE was enhanced to eliminate a lot of excess and unnecessary overhead when executed.

⎕SYS[25] Scalar Optimization Option Obsoleted

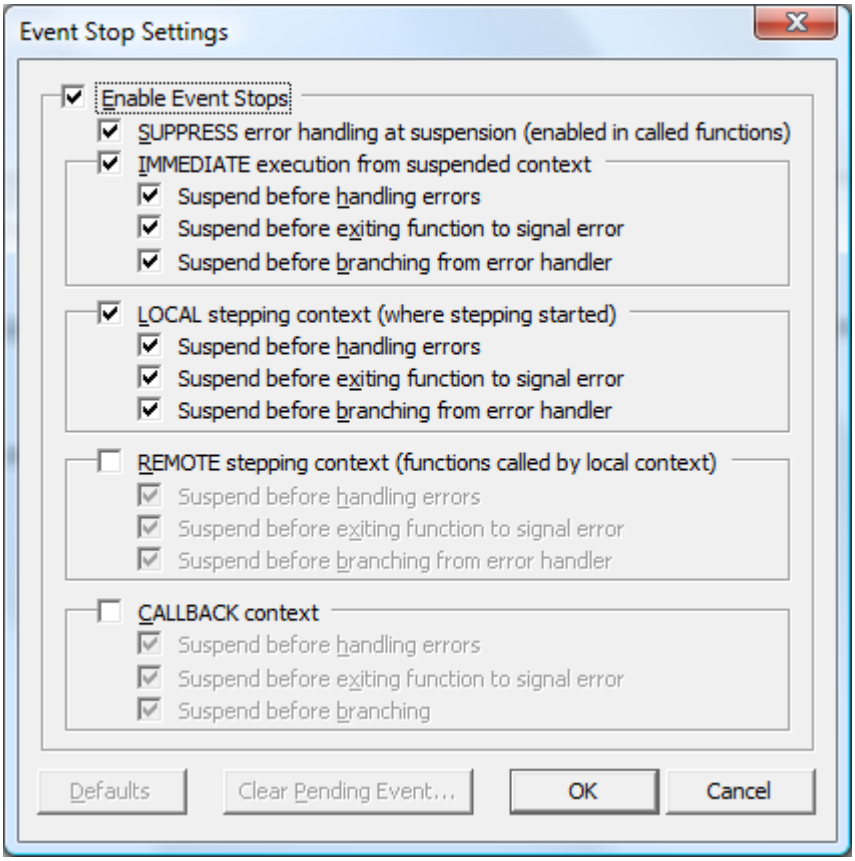
The new built-in scalar and array optimizations obsoleted this feature from prior versions. So enabling this option no longer affects in APL+Win.

Allow Tabs to appears in APL code without SYNTAX ERROR

Tabs are now treated syntactically the same as SPACEs when not in quotes. Inside quotes they have always been included into the string's value and that behavior remains unchanged. They only cause SYNTAX ERRORs when used to indent code or comments (which often happened as a result of pasting from another application that inserts tabs rather than spaces into the code).

New Debugging “Event Stop” Feature

The Event Stop feature makes it possible for developers to debug error handling code in ways that were previously difficult or impossible. At the core of this feature is a new Code Walker dialog named "Event Stop Settings" that contains the following set of checkbox options, arranged in a hierarchy, where subordinate items are disabled if the controlling item above them is not checked. This structure is shown below:



This dialog is bound to the F8 key, which can toggle the dialog display on and off (i.e., pressing F8 from the session displays the dialog, pressing F8 again when the dialog is already displayed hides it). You can also close the dialog via the OK button (or by pressing ENTER) or the Cancel button (or by pressing ESCAPE). There is also a new button on the CodeWalker toolbar that displays the dialog.

The top level "Enable Event Stops" option lets you enable or disable all other Event Stop features with a single click. When this option is checked, all the subordinate options below it are enabled, allowing them to be individually configured. When this option is disabled, all previous subordinate settings are preserved, but they are temporarily disabled. Unchecking this option restores the system to pre-version 10.0 behavior with respect to debugging of errors. Checking this option (the default) enables a standard subset of features (described later).

The "SUPPRESS error handling at suspension (enabled in called functions)" is one of the most important new debugging option. It suppresses execution of normal error handling at the immediate execution level when you are suspended (such as for debugging). This option is on by default. It prevents the long standing annoying problem of error handlers blowing the execution context away when they occur during suspension while debugging a problem. In the past the only way to avoid this was to reset `⎕ELX` to "`⎕DM`". Doing this was problematic however, since disabling the error handler could easily be forgetten, and running before the suspension with error handling disabled would change the behavior of the application. Furthermore, this trick did not work if you were suspended inside of a :TRY block. In that case, there was no way to prevent an error in immediate execution from triggering the error handling code, possibly blowing you out of the debugging context you may have worked hard to get into.

Consider the following set of functions that suspend with an ⎕ELX handler set to throw errors out of the context they occur in so they can be handled at the next higher level. This is a common occurence in APL error handling code, but one that makes debugging very difficult.

```
      ∇ sus
[1]    sus1
      ∇

      ∇ sus1
[1]    sus2
      ∇

      ∇ sus2;⎕elx
[1]    ⎕elx←'⎕error ⎕dm'
[2]    0 0⍴(1+⎕lc[1]) ⎕stop 'sus2'
[3]    ÷0
      ∇
```

Executing this code causes suspension on sus2[3] as shown below:

```
        sus
sus2[3]

        )SI
sus2[3] ⋆
sus1[1]
sus[1]
```

You might also have gotten into this condition, with an error handler poised to blow away the suspended context, by stepping into the code via Code Walker. However you get into this situation, it is precarious. One false move and you have to start over to set up the suspension condition again, in some cases blowing away hours of waiting for a bug to occur that suspends in such a case.

The following shows what happens on a pre-version 10.0 APL+Win system, or when the "SUPPRESS error handling at suspension" option is disabled. If you are trying to debug this code and cause an error, the ⎕ELX handler will execute and pop the error out of the context you were trying to debug. For example:

```
        1 2 3+4 5
LENGTH ERROR
        1 2 3+4 5
              ^
sus1[1] sus2
          ^

        )SI
sus1[1] ⋆
sus[1]
```

The LENGTH ERROR causes ⎕ELX handler to execute "⎕error ⎕dm" which blows the suspension at sus2[3] off of the ⎕SI stack and you end up suspended in the next level below, at sus1[1]. If the error handler was not localized strictly to sus2, then this error could have been progagated all the way back to the start of execution, or at least to the top level error trap in the application.

Then the "SUPPRESS error handling at suspension" option is enabled (as it is by default in APL+Win v10.0) the behavior is much more desirable. Error handling does not execute for errors that

occur in the immediate execution context above a suspension. Here's what happens instead, starting from the same suspension level as before:

```
        )SI
sus2[3] ⋆
sus1[1]
sus[1]
        1 2 3+4 5
LENGTH ERROR
        1 2 3+4 5
              ^
        )SI
sus2[3] ⋆
sus1[1]
sus[1]
```

The system simply prints out value of ⎕DM showing that a LENGTH ERROR has occured, and leaves you suspened again, exactly where you were before. In other words, it behaves as if ⎕ELX was set to "⎕DM". However, the ⎕ELX handler does not execute at all, in fact! Whatever ⎕ELX handler was in effect remains in effect, but its execution is simply suppressed in this context.

If the error occurs in a function called from this immediate context, error handler works normally, all the way back until the error reaches the immediate execution context above the suspension again. At that point, error propagation stops because the ⎕ELX handler does not execute. For example, consider the following pair of functions, and the effect of their exectuion while suspended at sus2[3]:

```
      ∇ makeerr
[1]    makeerr2
      ∇

      ∇ makeerr2
[1]    ⎕error'INTENTIONAL ERROR'
      ∇

        makeerr
INTENTIONAL ERROR
makeerr[1] makeerr2
              ^
        makeerr
        ^

        )SI
sus2[3] ⋆
sus1[1]
sus[1]
```

The error handler executes normally, propagating the error from makeerr2[1] back to makeerr[1] and then back into the immediate execution context. However, when it reaches the suspended immediate execution context, the handler does not fire, and the system suspends once again at sus2[3].

The next set of checkboxes on the Event Stop settings dialog control whether errors are intercepted before their handlers are executed, or before the final effect of a ⎕ELX error handler take effect (such as before exiting from the function where ⎕ERROR is called or branching to a different line of code). The top level options are:

IMMEDIATE execution from suspended context

LOCAL stepping context (where stepping started)

REMOTE stepping context (functions called by local context)

CALLBACK context

The IMMEDIATE context applies the same "immediate execution above a suspension" context that was discussed above. None of these options will apply if the SUPPRESS option is enabled, because in that case the error handler will not be executed. So these options only apply when the IMMEDIATE context option is disabled.

The LOCAL context applies to errors that occur in the same function where started a Code Walker step. Errors that occur in functions called by that starting context are controlled by the REMOTE context.

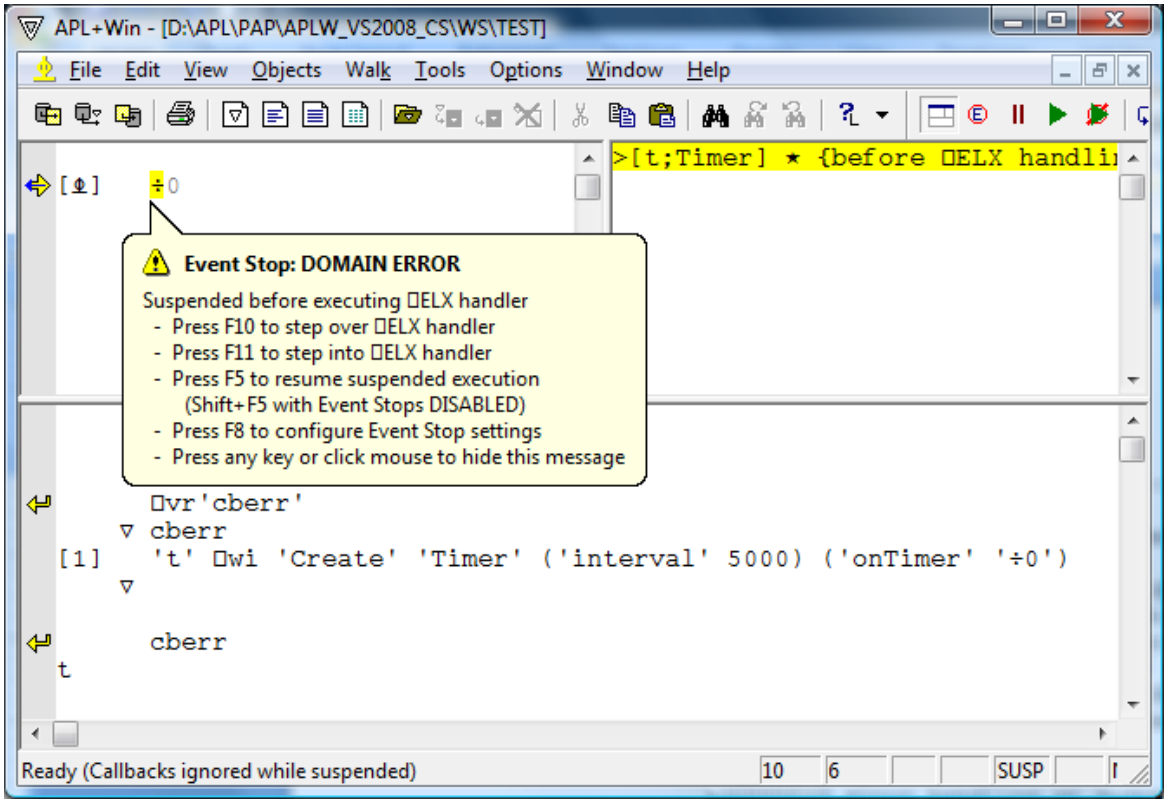
The LOCAL context is enabled by default whereas the REMOTE context is disabled by default. This means that when you are stepping in CodeWalker, the options below the LOCAL context will only apply if the error occurs locally, in the function that you are stepping in. If they occur in functions called by that function, this option will not apply, and those errors will not be trapped unless the corresponding REMOTE options are enabled.

Both the LOCAL and REMOTE options only apply when Stepping in Code Walker. Stepping is defined as doing operations such as F10 (Step Over), F11 (Step Into), F5 (Resume from suspension), etc. These options do not apply for normal execution. So these do not get in the way of error handling when an appliction is running normally. They only apply to the Code Walker debugging context, when you are stepping, or running after suspension. When you start execution via `⎕LX` when a workspace is loaded, or by typing a line and pressing ENTER in the session manager, you are not stepping in code walker and thus these error trapping options do not apply.

The CALLBACK context applies to the first level of execution in a callback. For example, this would apply to an error that occurs in the onTimer event handler. But it would not apply if that error occured in a function called by the onTimer event handler. The CALLBACK contexts is strictly at the boundary between the previous ŒSI level and the start of execution in the callback handler. The CALLBACK context is especially important for a couple of reasons. Sometimes errors occur in event handler callbacks that cannot be trapped or debugged otherwise. This allows you to stop at the point where the error is occuring, before handling, so you can see and fix the problem, rather than having it mysteriously print an error message in the APL session without being able to trap it. In addition, callbacks can signal errors and do branches in the context of the function level below them in the `⎕SI` stack. This can be the source of very hard to isolate bugs.

Event Stops allow you to stop when these kinds of events occur so you can debug and fix the problem.

The following example illustrates a case where enabling Event Stops in a callback context can be useful. I’ve created a timer object that fires an event that produces a DOMAIN ERROR every 5 seconds. Normally, this would be impossible to debug or even to stop without a lot of trouble. But with the CALLBACK event stop feature, the error is trapped before error handling is invoked so we can catch this in the act!



Each of the four debugging contexts is followed by a set of three subordinate options. The subordinate options are disabled if the controlling open above them is not checked:

Suspend before handling errors

Suspend before e&xiting function to signal error

Suspend before branching from error handler

The "Suspend before handling errors" option causes suspension of execution at the point where an error occurs, BEFORE its ⎕ELX, :CATCH, or :TRYALL handler is executed. This option allows you to suspend at the point immediately before error handling. In the case of :TRY /:CATCH blocks, this provides the ONLY mechanism you can use to catch and debug errors inside the body of the :TRY block before they are handled by the :CATCH clauses. This allows you to debug the error where it occurs rather than after execution has jumped into the :CATCH statements. This lets you see exactly where the error occurred, and explore its execution context before handling the error.

The "Suspend before exiting function to signal error" allows you to stop whenever ⎕ERROR is about to leave the calling function to signal the error in the context that called it. This allows you to stop to inspect the cause of the error that's about to be signalled before it leaves the context where it is being called. You cannot do that with traditional error trapping because the calling function will already have been exited by the time the error is signalled and the error handler is called. By then, it is too late to debug the cause of the conditions that lead up to the ⎕ERROR function being called.

The "Suspend before branching from error handler" allows you to stop before a ⎕ELX handler branches to another line to complete its error handling. In the CALLBACK context, this is slightly

different, but the same idea applies. In that case, the branch is trying to leave the callback context and branch into a function (perhaps a random function) in the context that was executing before the event handler was executed. This is actually a very dangerous situation that can lead to unpredictable behaviors in applications. In the CALLBACK context the branch might be occuring directly from the callback, or from an ⎕ELX handler that is executing due to an error that occured in the callback.

When execution suspends, a popup tooltip balloon is displayed with its arrow pointing at the suspended statement in the Code Walker window. It has a caption describing the error and the body tells what you can do about it. For example, if you were suspended at sus2[3] as in the previous example and pressed F10 to step to the next line, the DOMAIN ERROR that would occur there would be trapped before handling. This would cause the following balloon message to be displayed:

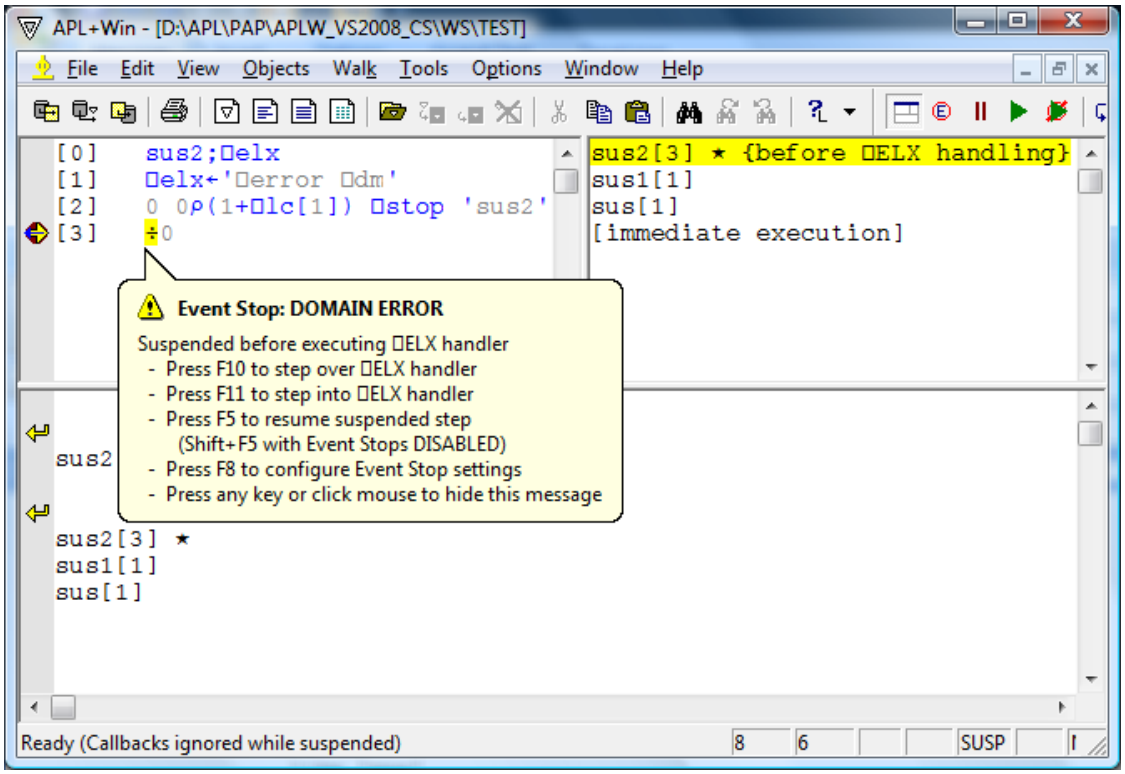
Event Stop: DOMAIN ERROR

Suspended before executing ⎕ELX handler

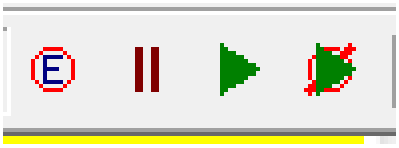
- Press F10 to step over ⎕ELX handler
- Press F11 to step into ⎕ELX handler
- Press F5 to resume suspended step

(Shift+F5 with Event Stops DISABLED)

- Press F8 to configure Event Stop settings
- Press any key or click mouse to hide this message



Also note the new buttons that appear on the Code Walker toolbar:



The "stop sign around the E symbol" is supposed to conjure up the Event Stop image in your mind.

There are now two different resume buttons that look like "play" buttons on recording appliences. F5 is hooked up to the Run/Resume function (as before) and the new Shift+F5 key is hooked up to the "play button overlaying the stop sign with a slash through it). This is suppposed to represent Run without Stopping (with Event Stops disabled). Anybody with better icon ideas is welcome to forward them at any time.

If the error occurs in a :TRY context, the message appears like this:

Event Stop: DOMAIN ERROR

Suspended before executing :CATCH handler

- Press F10, etc. to step to :CATCH statement
- Press F5 to resume suspended step

  (Shift+F5 with Event Stops DISABLED)
- Press F8 to configure Event Stop settings
- Press any key or click mouse to hide this message

If the error occurs in a :TRYALL context, the message appears like this:

Event Stop: DOMAIN ERROR

Suspended before executing :TRYALL handler

- Press F10, etc. to step to next line
- Press F5 to resume suspended step

  (Shift+F5 with Event Stops DISABLED)
- Press F8 to configure Event Stop settings
- Press any key or click mouse to hide this message

In the case of the pre-⎕ELX handling context, you can either step OVER the error handler (by pressing F10) or step INTO the error handler (by pressing F11). In the pre-:CATCH or :TRYALL contexts, pressing F10 or F11 will take you to the same place, the first :CATCH statement following the error or the next line in the :TRYALL context.

There are other balloon messages for other kinds of Event Stops. For example, The "Suspend before exiting function to signal error" displays a message such as this:

Event Stop: SOME ERROR

Suspended before exiting function to signal ⎕ERROR

- Press F10, etc. to signal ⎕ERROR
- Press F5 to resume suspended step

(Shift+F5 with Event Stops DISABLED)

- Press F8 to configure Event Stop settings
- Press any key or click mouse to hide this message

Whereas, the "Suspend before branching from error handler" dislays a balloon message like this:

Event Stop: Pending branch to line [4]

Suspended before branching

- Press F10, etc. to branch to target line
- Press F5 to resume suspended step

(Shift+F5 with Event Stops DISABLED)

- Press F8 to configure Event Stop settings
- Press any key or click mouse to hide this message

The Balloon tooltip was chosen because it is less invasive than a message box. You can continue debugging without having to close it. The regular debug stepping options work as usual in this context so you can either explore in immedite execution mode, or press F5, F10, F11, etc to continue in various ways. The balloon automatically disappear as soon as you press any key or click the mouse. It can be redisplayed again, but not directly. If there is a Event Stop pending when the Event Stop Settings dialog is closed, the system redisplays the pending event stop again, when the dialog is closed. So you can redisplay a vanished Event Stop balloon by pressing F8 twice in succession (this shows and hides the Event Stop Settings dialog and results in the pending Event Stop balloon being displayed again.

Whenever execution is suspended for an Event Stop that occured while stepping, the F5 key resumes execution, but in a slightly different way than usual. It resumes to completion of the current step that was in progress. If you were in the midst of a Run-to-Cursor operation, execution will resume running in that Run-to-Cursor mode and stop when it reaches that point. In other words, F5 finishes the current step that was in progress when the Event Stop interrupted it. This is convenient when you have REMOTE

context event stops enabled. In that case, an error might stop several levels deeper in the ⎕SI stack then were you started stepping. You can resume with F5 to continue back to the debugging context you were in when the Event Stop interrupted it.

You can start execution from the APL session by pressing F5 rather than ENTER to start execution. However, note that in that case, since execution starts from the immediate execution context, the LOCAL option will not apply to anything other than errors being throw back to that level. You must enable the REMOTE option in order to trap errors that occur when starting execution from the command line with F5.

The Shift+F5 key (or corresponding menu and toolbar options) can be used to resume execution after suspension with Event Stops temporarily suppressed. This allows you to resume after a suspension without allowing any subsequent Event Stops until that execution segment comes to an end. This emulates what would have happened if execution had started from the command line via the ENTER key. Note that Shift+F5 can also be used to start execution from the command line (rather than pressing ENTER). That will suppress Event Stop during that execution.

But it also suppreses event stops for non-stepping contexts such as the CALLBACK and IMMEDIATE contexts.

The default settings are chosen to be as uninvasive as possible. They fix the problem with error handlers blowing the execution context during suspensions (i.e., the SUPPRESS mode is on by default). They also catch errors that occur in the LOCAL stepping context. Those are the only options that are on by default. The REMOTE context is off to avoid trapping of possibly normal error conditions that get properly handled in functions called by the stepping context. If you want to trap errors at the point of origin in a called function you can enable the REMOTE context to do that. Similarly, errors may occur in callbacks that do not affect program execution and are basically benign. So by default CALLBACK error trapping is off also. However, if you have a problematic error occuring in a callback handler, you can easily enable this option to catch it.

In addition to the balloon messages displayed when Event Stops occur, they are also indicated in the ⎕SI display as an extended form of suspension. Normal suspensions are shown with a star (*) following the suspended function's name and line number (or other context indicator such as an execute or execute ⎕ELX prefix). So a normal)SI display looks like this for a suspended function:

```
        )SI
sus2[3] ⋆
sus1[1]
sus[1]
```

However, when suspended for an Event Stop, this suspension notation is enhanced with a curly-brace enclose suffix such as this:

```
        )SI
sus2[3] ⋆ {before ⎕ELX handling}
sus1[1]
sus[1]
```

If we step into the error handler and continue execution, we'll eventually get to the point where an error is signalled. The pending Event Stop for that will appear in ⎕SI like this:

```
        )SI
⍎⎕ELX ⋆ {before ⎕ERROR exit}
sus2[3] ⋆
```

```
sus1[1]
sus[1]
```

If the error handler is trying to branch out of an ⎕ELX handler, the ⎕SI will appear like this:

```
⍎⎕ELX ⋆ {before branching}
elxerr[3] ⋆
```

These extensions are intended to "look like" regular suspensions to any utility functions that programmatically inspect the ⎕SI stack. Most such utility functions check for suspension levels by simply finding rows of ⎕SI that contain a "*" character. Those functions will continue to work without fication. This was a superior solution to adding a new ⎕SI level with an unexpected format that would have more likely been confusing to utility code.

The Event Stops feature is expected to be substantially enhanced in a future version of the system to include error filtering so that stops can be triggered to occur only for certain error or only in certain functions. In addition, it is planned to replace the existing ⎕WATCHPOINTS feature with integration into the Event Stop system. Because the representation of advanced Event Stop filtering has not yet been fully worked out, the present version of the system does not provide a system variable that contains the event stop settings. It is planned this will be introduced in the future as ⎕ES. Because of this, there is no way in APL+Win v10.0 to preserve or programmatically change Event Stop settings between sessions. They are reset to the default at the start of every APL session.

However, the default settings make them both useful and "safe". They are "safe" in the sense that the default settings cannot cause unexpected suspensions in production applications. They only affect the immediate execution and local stepping context. Neither of those contexts can exist in a production application. It would actually be a good idea to enable CALLBACK error trapping by default. However, this option is off by default because we did not want to have an effect in production applications.

Allow :RETURN argument to specify result value

This enhancement extends the syntax of the :RETURN and the :RES clause of :RETURNIF statements and adds a new ⎕NOVALUE system variable. For example:

```
:RETURN

:RETURN result

:RETURN ⎕NOVALUE

:RETURNIF condition

:RETURNIF condition :RES result

:RETURNIF condition :RES ⎕NOVALUE
```

The :RETURN statement now allows an argument to specify the result value. For example, if local variable Z is declared as the result for function Foo as shown below, then in the past you had to do the following to set the value of Z and return it from the function:

```
[0]  Z ← Foo;Loc1;Loc2;...
     ...
[8]  Z ← X Goo Y
[9]  :RETURN
```

```
      ...
```

But now you can do it like this:

```
[8]   :RETURN X Goo Y
```

Note that in the second case when :RETURN specifies an argument, it is not necessary to assign a value to result variable Z. The argument to the :RETURN statement is used as the function result value regardless of whatever value (if any) was previously assigned to Z. In fact, you can use a :RETURN statement with an argument to return a value from a function that does not declare a result variable!

When an argument is not used with :RETURN the behavior remains as in the past. The current value (if any) bound to the declared result variable for the function is returned by the function. But when an argument is specified, the value of the argument expression is used as the result value regardless of the value of the declared result variable or even the existence of a result variable declaration.

If the value expression does not return a value, such as calling some function that doesn't set a result value, then the function executing the :RETURN statement does not return a value either. The new NO-VALUE system variable ⎕NOVALUE can be used to state this explicitly like this:

```
:RETURN ⎕NOVALUE
```

⎕NOVALUE is available as a way to explicitly return no value, overriding whatever value might have been previously set for the declared result variable. It gives a VALUE ERROR when used in ALL other contexts. In other words, it is just like referencing a undefined name, in all contexts other than as return argument.

If the statement above is executed, any declared result variable is ignored and the function does not return a value. If Z is the declared result variable, then the statement above is functionally equivalent to the statement below:

```
0 0⍴⎕ERASE 'Z'
:RETURN
```

As with the :THEN keyword, the parenthesis shown in the first case above are only necessary if the leading colon (:) of the :RETURN keyword could be confused with statement label syntax. So you can code this:

A > 10 :RETURN X

but the following would be confused with statement label A followed by statement "RETURN Y" which might or might not be detected as an error depending upon how (or if) the RETURN and Y names are defined in the workspace:

A :RETURN Y

So parens would be required like this:

(A) :RETURN Y

or like this:

(A :RETURN Y)

When a condition prefix is specified the :RETURN keyword MUST be followed by an result value expression. If you don't want to return a value you can use ⎕NOVALUE as the argument.

When the :RETURN statement specifies an explicit result value the declared result variable (if any) is assigned that value. If there isn't a declared result variable, then the result value is stored in a special "hidden" result variable that is implicitly localized in the function header (but not shown).

Note that you CANNOT use the explicit result form of :RETURN with a ⎕-name as the declared result variable!

In the future, if we allow multiple result variables to be declare, you will not be able to use them with an explicit :RETURN value. That's because there isn't any way to assign an arbitrarily shaped result value to a vector-shaped strand of names in such a way that they can be returned as the result. The :RETURN statement with an argument will only be valid to use with functions that declare ONE or NONE result variables.

New :RETURNIF, :LEAVEIF, and :CONTINUEIF Control Structure Statements

The new :RETURNIF, :LEAVEIF, and :CONTINUEIF statements allow conditional return from a function and exit/continue from a loop. These statements are shortcuts for an :IF statement containing a corresponding :RETURN, :LEAVE, or :CONTINUE. The following statements:

```
:IF condition
      :RETURN
:END

:IF condition
      :LEAVE
:END

:IF condition
      :CONTINUE
:END
```

can be replaced by the much more compact versions below:

```
:RETURNIF condition

:LEAVEIF condition

:CONTINUEIF condition
```

In addition, the following :RETURN with result value argument:

```
:IF condition
      :RETURN result
:END
```

can be replaced by the following single statement:

```
:RETURNIF condition :RES result
```

Because there isn't an argument option for :LEAVE or :CONTINUE they don't support a :RES clause.

NOTE: The new :RETURNIF statement with :RES clause replaces the :RETURN statement with conditional prefix. The :RETURN statement without a prefix allows an optional result value following the :RETURN keyword. But the conditional prefix no longer works and this previous syntax:

```
Condition :RETURN Result
```

Has been replaced by the following:

```
:RETURNIF Condition :RES Result
```

Implement :TEST,)TEST, and ⎕TEST

The :TEST statement controls a block of statements that are not run during normal program execution. Any :TEST blocks in your code essentially disappear from normal program execution as if they were comments. They are not reachable and result in a DESTINATION ERROR if you try to branch into them under when running the function normally. The :TEST statement has this block structure:

```
:TEST
      init statements
:PASS
      pass statements
:FAIL [error]
      fail statements
:ENDTEST
```

When a test block begins execution, the locals and labels declared for the function that contains it are localized and labels set to their declared line values, like a normal execution of the function. However, arguments are not assigned (as if the function were invoked niladically). And if you assign a result to the result variable, its value is discarded upon return from the test. A future version may allow local variables to be declared as part of the test block such as this:

```
:TEST;A;B;C
      init statements
:PASS
      pass statements
:FAIL [error]
      fail statements
:ENDTEST
```

Labels can not be declared INSIDE :TEST blocks (but labels can be declared in the normal code outside of test blocks). You get a `DESTINATION ERROR` if you try to branch from inside a test block to outside (or vice versa).

Functions that are being tested display in `⎕SI` with a "{Test}" suffix. `⎕SI` displays as:

```
Foo[27] {Test}
```

when testing Foo[27]. The {Test} extension is intended to be transparent to utility functions that inspect `⎕SI`. Any such function that parses the function[line] code should continue to work normally in most cases (assuming they don't also check for specific patterns following the closing bracket following the line number. Most such utilities only look past that point in an `⎕SI` line by scanning for lines containing "*"

as a suspension indicator. So hopefully, the {Test} extension will not create too many problems for existing utility code.

Testing is invoked via the)TEST command or ⎕TEST function. Their argument allow pattern specifications for the names of functions to be selected for testing. The patterns may contain * and ? wildcard characters similar to those allowed in the :CATCH clause argument.

```
)TEST Foo Goo ⋆OO
```

or

```
⎕TEST 'Foo Goo ⋆ish'
```

Both lines above test functions Foo, Goo, and all functions that end with "ish" in their names. Of course, only those functions matching the pattern that contain :TEST blocks are actually tested.

Any name or pattern prefixed by an UP-ARROW (↑) prefix is executed such that test failures cause an Event Stop to occur.

The prefix applies only to the next function (or set of functions if a wildcard pattern is used).

When tests are running, ⎕ELX is set to '⎕THROW ⎕DM' so that unhandled errors that occur in called functions are propagated back to the :TEST context if they are not handled by the called functions. For example, the following allows you to debug the code during tesing of Foo:

```
)TEST ↑Foo
```

Example execution of)TEST on the <test1> function is shown below:

```
     ∇ test1;x;a;b;c
[1]   :test
[2]     a←10
[3]     b←20
[4]     c←30
[5]   :pass
[6]     c←c
[7]   :pass
[8]     :assert 10=10
[9]   :fail
[10]    :assert 10=20
[11]  :fail
[12]    ÷0
[13]  :fail 'INDEX ERROR'
[14]    10 20[3 4]
[15]  :end
     ∇

       )test test1

C:/ProgramData/APL+Win/Log/Test/Test-20100812115228791.log

                    ------------------------------------

{{{{{{{{{{{{ Test Start 2010-08-12 11:52:28.791 UTC {{{{{{{{{{{{

                  ------------------------------------
```

```
>>> test1[1]       TEST starte
test1[1]       init succes
test1[5]       case succes
test1[7]       case succes
test1[9]       case succes
test1[11]      case succes
test1[13]      case succes
test1[15]      test finished
     ------------------------------------

}}}}}}}}}}}}} Test Stop  2010-08-12 11:52:32.258 UTC }}}}}}}}}}}}}

     ------------------------------------

Test Log File C:/ProgramData/APL+Win/Log/Test/Test-20100812115228791
.log

Functions     1
Test Blocks   1
Test Cases    5
Used :IFTEST  0
Init Errors   0
Case Errors   0
Skipped Cases 0

     ------------------------------------
```

The following test includes failures:

```
     ∇ test2 fail;xx;a;b;c

[1]   ⍝ Normal Code:
[2]   :if fail
[3]     test2a ⍝ calls test2b which calls test2c which fails
[4]   :else
[5]     a←⍳10  ⍝ does not fail
[6]   :end
[7]
[8]   ⍝ Test Code:
[9]   :test
[10]    a←10
[11]    b←20
[12]  :pass
[13]    test2 0 ⍝ NO error is success in :PASS clause
[14]  :pass
[15]    test2 1 ⍝ error is failure in :PASS clause
[16]  :fail
[17]    test2 1 ⍝ error is success in :FAIL clause
[18]  :fail
[19]    test2 0 ⍝ NO error is failure in :FAIL clause
[20]  :end
     ∇

     ∇ test2a
[1]
[2]   :iftest
[3]     xx←⍳10
[4]   :endif
```

```
      [5]
      [6]    test2b
           ∇

           ∇ test2b
      [1]    ⎕throw'INTENDED ERROR'
           ∇

      )test test2

C:/ProgramData/APL+Win/Log/Test/Test-20100812122153616.log

                  ---------------------------------------

{{{{{{{{{{{{{ Test Start 2010-08-12 12:21:53.616 UTC {{{{{{{{{{{{{

                  ---------------------------------------

              >>> test2[9]       TEST started

             test2[9]       init success
             test2[12]      case success

!!! test2a[2]      used :IFTEST @ test2a[2] test2[3] test2[15] {Test}

***FAILURE*** test2[14]      case FAILURE: Unexpected Error

      INTENDED ERROR

      test2b[1] ⎕throw'INTENDED ERROR'

                         ^

            test2a[6] test2b

                         ^

            test2[3]   test2a ⍎ calls test2b which calls test2c which fails
                     ^

            test2[15]   test2 1 ⍎ error is failure in :PASS clause

                           ^


        !!! test2a[2]      used :IFTEST @ test2a[2] test2[3] test2[17] {Test}

                               test2[16]      case success

            ***FAILURE*** test2[18]      case FAILURE: Expected error did not occur

                               test2[20]      test finished

                               ---------------------------------------

            }}}}}}}}}}}}} Test Stop  2010-08-12 12:21:53.686 UTC }}}}}}}}}}}}}

                               ---------------------------------------


      Test Log File C:/ProgramData/APL+Win/Log/Test/Test-20100812122153616.log
```

```
Functions     1
Test Blocks   1
Test Cases    4
Used :IFTEST  2    ★★★WARNING★★★ remove :IFTESTs when no longer needed
Init Errors   0    Case Errors   2    ★★★FAILURE★★★
Skipped Cases 0

        ------------------------------------
```

The following example has a failure in the initialization section and therefore skips execution of all its :PASS and :FAIL cases:

```
      ∇ test3;x;a;b;c

[1]   :test
[2]     a←10
[3]     b←20
[4]     c
[5]   :pass
[6]     c
[7]   :pass
[8]     c←30
[9]   :pass
[10]    :assert 10=20
[11]  :fail
[12]    :assert 10=20
[13]  :fail
[14]    ÷0
[15]  :fail
[16]    ÷0
[17]    x←10
[18]  :fail 'INDEX ERROR'
[19]    ⎕error'INDEX ERROR'
[20]  :fail 'INDEX ERROR'
[21]    ⎕error'INDEX ERROR'
[22]    c←10
[23]  :fail 'DOMAIN ERROR'
[24]    ⎕error'INDEX ERROR'
[25]  :fail 'DOMAIN ERROR'
[26]    c←10
[27]  :fail
[28]    c←10
[29]  :end
      ∇

  )test test3

C:/ProgramData/APL+Win/Log/Test/Test-20100812115510119.log

        ------------------------------------

{{{{{{{{{{{{{ Test Start 2010-08-12 11:55:10.119 UTC {{{{{{{{{{{{{

        ------------------------------------

  >>> test3[1]       TEST started

★★★FAILURE★★★ test3[1]       init FAILURE: Unexpected Error
```

```
      VALUE ERROR

      test3[4]   c

                 ^

      ***skipped*** test3[5]       case SKIPPED: due to init failure

***skipped*** test3[7]       case SKIPPED: due to init failure

***skipped*** test3[9]       case SKIPPED: due to init failure

***skipped*** test3[11]      case SKIPPED: due to init failure

***skipped*** test3[13]      case SKIPPED: due to init failure

***skipped*** test3[15]      case SKIPPED: due to init failure

***skipped*** test3[18]      case SKIPPED: due to init failure

***skipped*** test3[20]      case SKIPPED: due to init failure

***skipped*** test3[23]      case SKIPPED: due to init failure

***skipped*** test3[25]      case SKIPPED: due to init failure

***skipped*** test3[27]      case SKIPPED: due to init failure

             test3[29]      test finished

             ---------------------------------------

}}}}}}}}}}}}} Test Stop  2010-08-12 11:55:10.269 UTC }}}}}}}}}}}}}

             ---------------------------------------

Test Log File C:/ProgramData/APL+Win/Log/Test/Test-20100812115510119.
log

Functions     1
Test Blocks   1
Test Cases    11
Used :IFTEST  0
Init Errors   1    ***FAILURE***
Case Errors   0
Skipped Cases 11   ***skipped***

             ---------------------------------------
```

The :IFTEST statement defines a block of statements that are conditionally execute depending upon whether the system is running in test or normal mode. It is stated like this:

```
:IFTEST
      test statement(s)
:ELSE
      non-test statement(s)
:ENDIFTEST
```

This conditionally executes code in either the test clause when a function is executed in test mode. If an :ELSE it coded, the non- test statements are executed when not run in test mode.

This is similar to the :IFDEBUG statement except that the :IFTEST function does not have zero overhead when testing is not being done.

The test statement's don't executed in normal execution. But there is a small overhead each time the :IFTEST statement is called. So it is not as optimized as :IFDEBUG in release mode. That :IFTEST is a temporary block of code, which is intested to be for "bridging code" during development and should be eliminated once development is done. When you run a)TEST containing :IFTEST blocks, they are listed in the)TEST results as a reminder to remove the tests once they are no longer needed.

```
      ∇ test5;xx;a;b
[1]     ⍝ Normal Code:
[2]     test5a
[3]
[4]     ⍝ Test Code:
[5]     :test
[6]       a←10
[7]       b←20
[8]     :pass
[9]       test5
[10]    :pass
[11]      test5a
[12]    :fail
[13]      ÷0
[14]    :fail 'INDEX ERROR'
[15]      1 2 4[5 6]
[16]    :end
      ∇

        )test test5

C:\ProgramData\APL+Win\Log\Test\Test-20100901201146186.log

                ---------------------------------------
{{{{{{{{{{{{ Test Start 2010-09-01 20:11:46.186 UTC {{{{{{{{{{{{

                ---------------------------------------

          >>> test5[5]       TEST started
              test5[5]        init success

***FAILURE*** test5[8]       case FAILURE: Unexpected Error

VALUE ERROR

test5[2] test5a

         ^

test5[9] test5

         ^

***FAILURE*** test5[10]      case FAILURE: Unexpected Error

VALUE ERROR
```

```
     test5[11] test5a
               ^
                   test5[12]      case success
                   test5[14]      case success
                   test5[16]      test finished
                   ------------------------------------
     }}}}}}}}}}}}} Test Stop  2010-09-01 20:11:46.269 UTC }}}}}}}}}}}}}
                   ------------------------------------
Test Log File C:\ProgramData\APL+Win\Log\Test\Test-20100901201146186.log
     Functions     1
     Test Blocks   1
     Test Cases    4
     Used :IFTEST  0
     Init Errors   0
     Case Errors   2    ★★★FAILURE★★★
     Skipped Cases 0
                   --------------------------------------
```

New :EX Control Structure Statement

The :EX statement is used in cascading decision statements (such as:AND/OR extensions of :IF, :WHILE, etc.) statements as a place to compute something before taking the next decision. The :EX clause is not allowed in ICS expressions for version 10.0 but it may be supported in the future.

For example:

```
:if cond1
:and cond2
:ex calc1
:ex calc2
:and cond3
     true statement(s)
:else
     false statement(s)
:end
```

The values you calculate in a :EX clause may be needed as inputs to the next conditional :AND/OR test in the sequence. Perhaps you need to compute a Boolean vector, V, that's used in more than one place in the conditional expression that follows, but the value of the V is not directly decisional. It is just needed in order to make the next decision. However, because it is used twice in the decisional logic, you would

like to avoid calculating the same value twice in that expression. Being able to assign it to an intermediate variable is very convenient. The pair of equivalent examples above make decisions based on cond1 and cond2, then they "calculate some intermediate values" in the pair of :EX expressions that may be needed for the cond3 decision. The :EX statement allows you to defer calculation of a value until you get to the point in the decision logic where it is going to be needed, but without requiring an additional logical level (such as a nested pair of :IF statements) in order to express it).

Allow ⎕DM Localization and Assignment

⎕DM can now be localized to preserve its value upon exit from a function. This allows the state of ⎕DM to remain unchanged upon exit from a function that may have produced errors and changed the state of ⎕DM during its execution. ⎕DM can also be assigned a character vector or scalar value. If the value being set contains any embedded ⎕TCNUL, it is truncated at that character.

Allow Scope Left Argument to ⎕AT and ⎕SIZE

⎕AT can now be called like this:

```
      1 ⎕AT Names
      2 ⎕AT Names
Scope 1 ⎕AT Names
Scope 2 ⎕AT Names
```

where Scope is the same as the Scope argument that can be used with ⎕VR, to specify the ⎕SI scope at which to look for the value.

⎕SIZE can be called monadically or dyadically with a simple left argument like this:

```
      ⎕SIZE Names
Scope ⎕SIZE Names
```

For example:

```
      2 ⎕at 'sus'
2010 8 19 23 46 41 0
      ⎕size 'sus'
268
      sus
sus2[4]
      )SI
sus2[4] ⋆
sus1[1]
sus[1]
      2 ⎕at 'sus'
0 0 0 0 0 0 0
      0 2 ⎕at 'sus'
2010 8 19 23 46 41 0
      1 2 ⎕at 'sus'
0 0 0 0 0 0 0
      ⎕size 'sus'
64
      0 ⎕size 'sus'
268
```

```
      1 ⎕size 'sus'
64

    ∇ sus
[1]   sus1
    ∇

    ∇ sus1
[1]   sus2
    ∇

    ∇ sus2;⎕elx;sus
[1]   sus←⍳10
[2]   ⎕elx←'⎕error ⎕dm'
[3]   0 0⍴(1+⎕lc[1]) ⎕stop 'sus2'
[4]   ÷0
    ∇
```

Clean Orphans on Every)LOAD,)COPY, and)SAVE Operations

Orphans (abnormal objects) are automatically cleaned every time a workspace is loaded, copied, or saved. This is done silently, without any notifications to the user when orphans are discovered. Orphan cleaning can be suppressed for)LOAD,)COPY, and)SAVE via the [Config]NoOrphanClean parameter. The default value (0) does not suppress orphan cleaning. This parameter may contain the sum of the following values to selectively suppress orphan cleaning. Normally, the only option that makes sense to suppress is cleaning of orphans on)SAVE so that workspaces in which orphans are being generated can be saved for inspection by APL+Win developers. The available options are:

0 Do not suppress orphan cleaning
1 Suppress orphan cleaning for)SAVE
2 Suppress orphan cleaning for)LOAD
4 Suppress orphan cleaning for)COPY

Suppress "Virtual Memory Over Commit" Dialog Box

Add INI file parameter [Config]VirtualCommitPrompt=1 or 0 to control what happens when virtual memory cannot be committed. This allows server applications to gracefully execute without displaying a message box to a non-existent user. The default has been changed so that the memory over-commitment dialog is not displayed by default. It is only displayed when this option is set to 1. When this option is set to 0 (the default), rather than prompting the user in memory overcommit conditions, we instead signal WS FULL error. If the commit failure occurs when trying to load a workspace, the error message is WS TOO LARGE rather than WS FULL.

This is a very unusual error condition, and would be preceded normally by a huge degradation in system performance as less and less memory becomes available on the system overall. By returning an error rather than prompting, we completely avoid the problem of hanging server applications, as well as mystifying users with a cryptic prompt that would often be more confusing than helpful.

When [Config]VirtualCommitPrompt=1 is set, the system crashes with a "WS FULL: UNABLE TO COMMIT VIRTUAL MEMORY" message ONLY after the message box has been displayed and if the user presses its CANCEL button indicating they want to terminate APL+Win rather than wait for virtual memory to become available. If the user clicks the OK button, APL+Win will resume trying to

commit the virtual memory, and will redisplay the message box again if it cannot do so. This cycle continues until the memory becomes available or they give up and presses the CANCEL button, leading to the crash that is correctly described already. Recovery applications can search for this string in the log file as a signal to go gently with respect to memory usage in the restarted application.